IERG4330/ IEMS5730

Apache Flink

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

Most of the Slides in this talk have been adapted from the following sources:

- Kostas Tzoumas, TU-Berlin, "Analyzing and Linking Big Data with Stratosphere," June 2012.
- Kostas Tzoumas, Co-founder and CEO of dataArtisans, Apache Flink Committer, "Apache Flink", Jan 2015.
- Stephen Ewen, Co-founder and CTO of dataArtisans, Apache Flink Committer, "Apache Flink", Jan 2015.
- Christoph Boden, "Introduction to Flink," Technologie-Workshop Big Data, FZI Karlsruhe, June, 2015.
- Kostas Tzoumas, Co-founder and CEO of dataArtisans, Apache Flink Committer, "Apache Flink: State of the Union and What's Next", Strata+Hadoop World, NYC, Sept 2016.
- Tzu-Li Tai of dataArtisans, "Stateful Stream Processing with Apache Flink," Flink Meetup@Idealo GmbH, June 2017
- Prof. Volker Markl, BBDC, TU-Berlin, "Big Data: Challenges and some Solutions: Stratosphere, Apache Flink and Beyond," Nov. 2017
- Stephen Ewen, CTO of dataArtisans, "Apache Flink and Stateful Stream Processing," Qcon London, Mar 2018.
- P. Nowojski, "Apache Flink: Better, Faster & Uncut," Big Data Technology Summit, Warsaw, 2018
- N. Kruber of dataArtisans, "What's new in Stateful Stream Processing with Apache Flink 1.5 and beyond," Flink Forward, SF, June 2018
- A. Krettek, Till Rohrmann, Co-founders and Engineering Managers of dataArtisans, "The Past, Present and Future of Apache Flink," Flink Forward, Berlin, Sept 2018.
- Timo Walther, "Introduction to SQL on Apache Flink," Flink Forward, Berlin, Sept 2018.
- Fabian Hueske, "SQL on Data Streams," Flink Forward, Berlin, Sept 2018.
- Timo Walther, "Flink's Table API & SQL Ecosystem," Flink Forward, Berlin, Sept 2018.
- A. Zagrebin, of dataArtisans, "Introduction to Apache Flink," Nov. 2018.
- Flink Forward,Beijing, Dec 2018,
- Till Rohrmann, Engineering Lead at dataArtisans, "Apache Flink 1.7 and Beyond," Flink Forward, Beijing, Dec 2018.
- Fabian Hueske, "Apache Flink SQL in Action," Feb 2019.
- Fabian Hueske, Vasiliki Kalavri, Streaming Processing with Apache Flink (Early Access Edition), 1st Edition to be published by O'Reilly Publishers in April 2019.

Where does Apache Flink come from ?

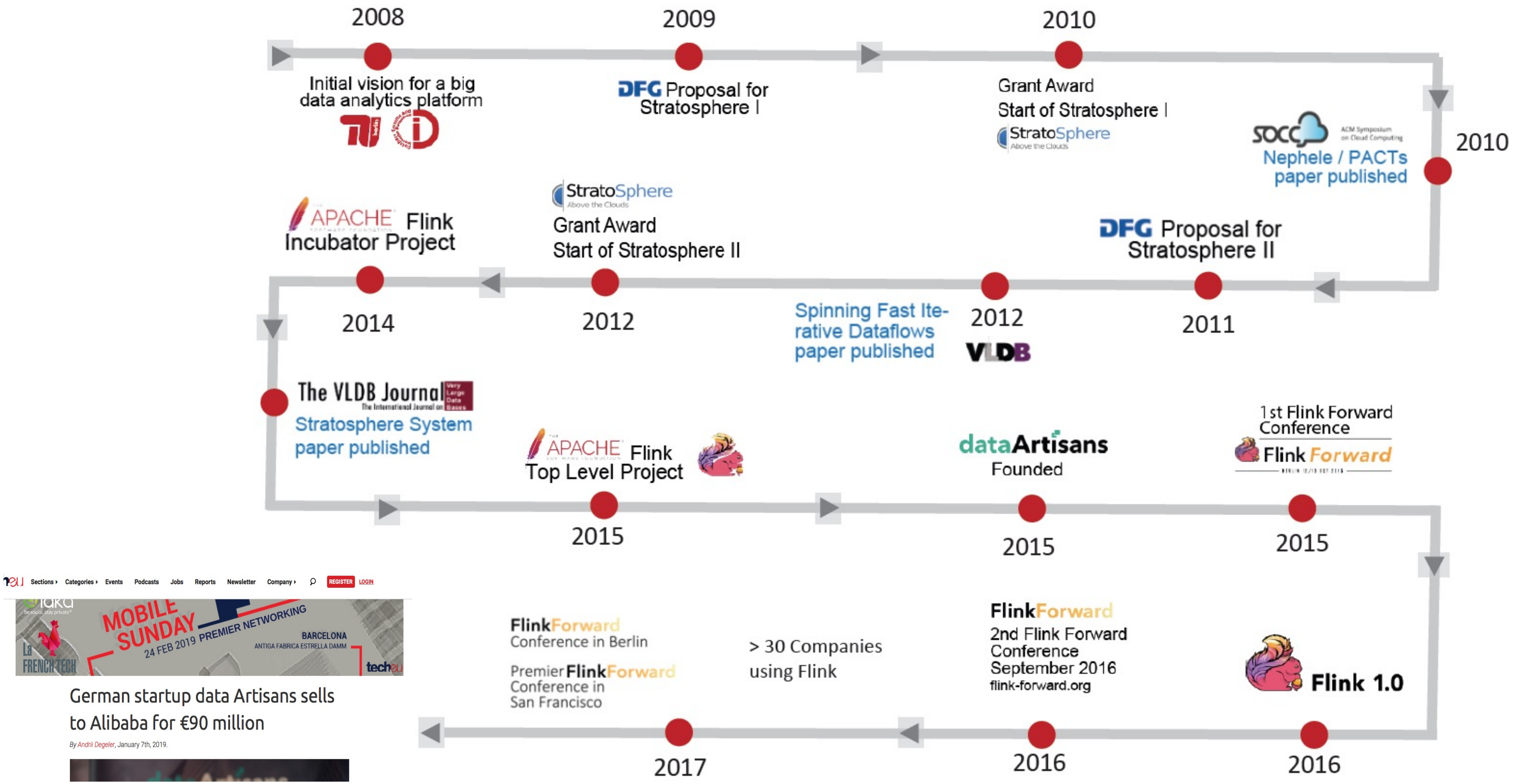
It all started in 2014

2009 - 2014

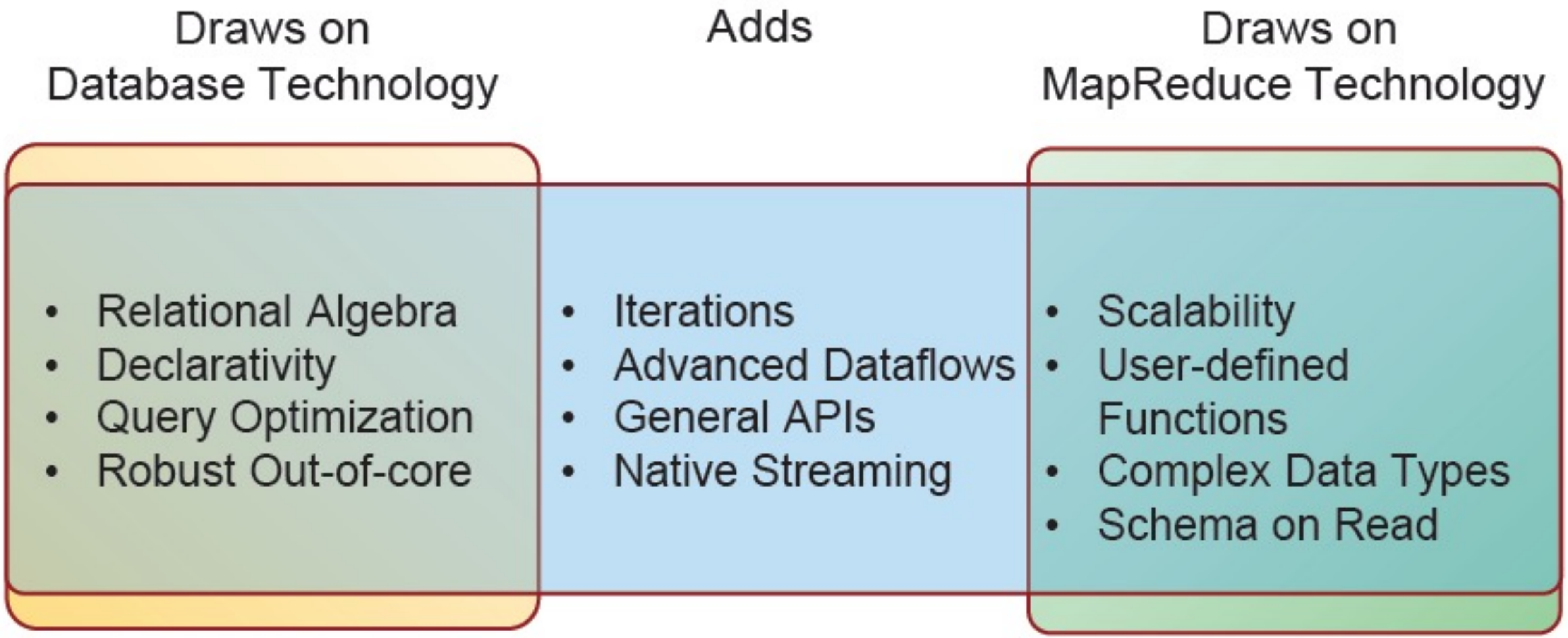
since 2014

- Batch processor on top of streaming runtime
- First Apache Flink 0.6.0 release August 2014

Evolution Timeline of Flink



Stratosphere: General Purpose Programming+Database Execution



A. Alexandrov, D. Battré, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, D. Warneke: Massively Parallel Data Analysis with PACTs on Nephele. PVLDB 3(2): 1625-1628 (2010)

D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke: Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. SoCC 2010: 119-130

A. Alexandrov, R.Bergmann, S. Ewen, et al: The Stratosphere platform for big data analytics. VLDB J. 23(6): 939-964 (2014)

Stratosphere 0.4

Pact API (Java) | DataSet API (Scala)

Stratosphere Optimizer

Stratosphere Runtime

Local | Remote

Batch processing on a pipelining engine, with iterations …

Eventually becomes Flink

Data Stream Processing
realtime results from data streams

Batch Processing
process static and historic data

Event-driven Applications
data-driven actions and services

Stateful Computations Over Data Streams

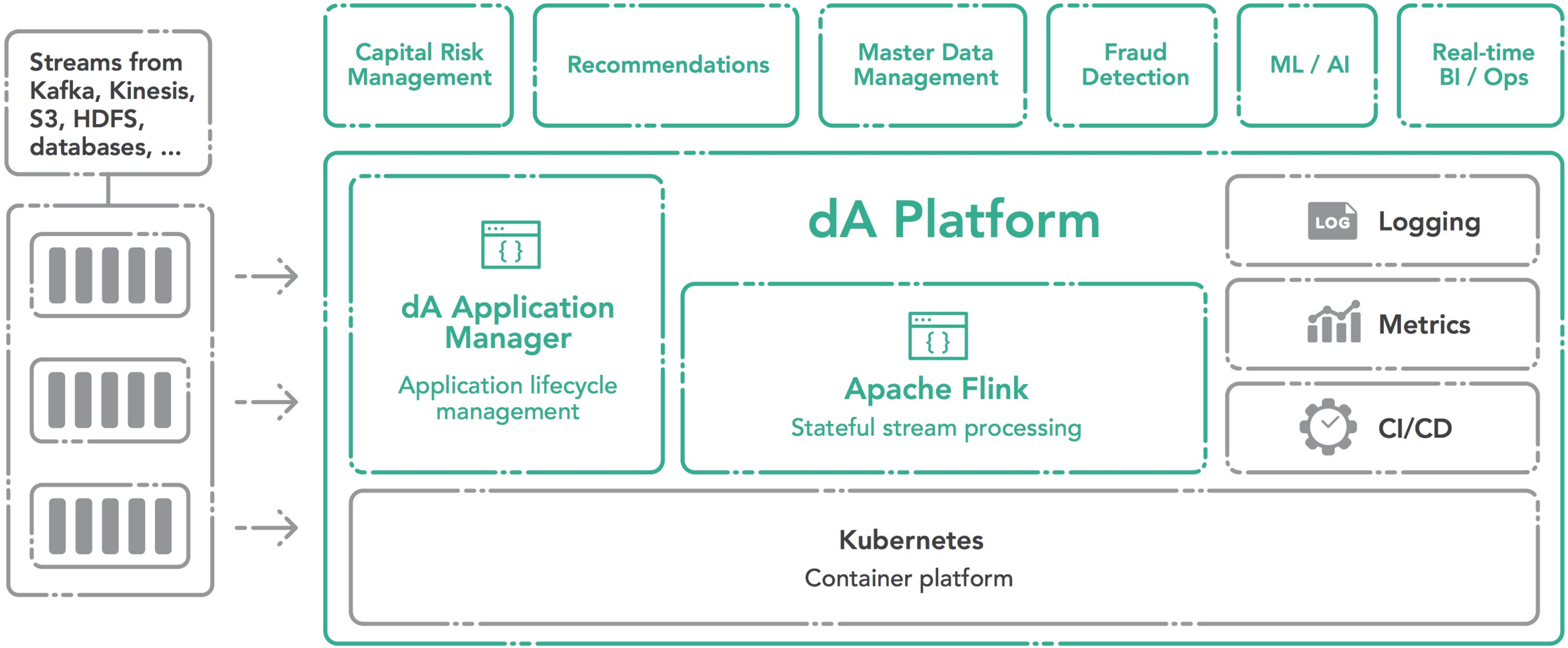
dataArtisans

dA
PLATFORM

Original creators of
Apache Flink®

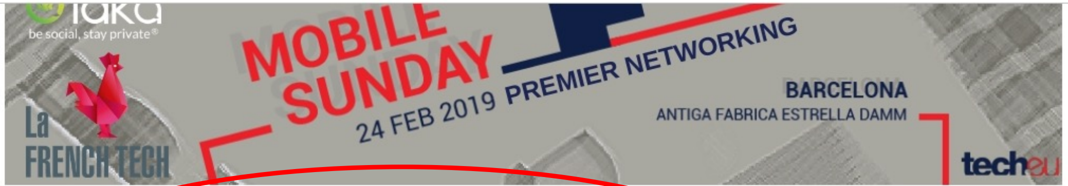
dA Platform 2
Stream Processing for the
Enterprise

dA platform



Streams from Kafka, Kinesis, S3, HDFS, databases, ...
Capital Risk Management
Recommendations
Master Data Management
Fraud Detection
ML / AI
Real-time BI / Ops
dA Platform
dA Application Manager
Application lifecycle management
Apache Flink
Stateful stream processing
LOG
Logging
Metrics
CI/CD
Kubernetes
Container platform

tech.eu Sections ▸ Categories ▸ Events Podcasts Jobs Reports Newsletter Company ▸ REGISTER LOGIN



German startup data Artisans sells to Alibaba for €90 million

By *Andrii Degeler*, January 7th, 2019.

Chinese e-commerce giant Alibaba Group has reportedly acquired Berlin-based data Artisans.

Kostas Tzoumas
@kostas_tzoumas
Follow

Super excited to announce our new name! data Artisans ==> Ververica. Read about it here: ververica.com/blog/introduci… and follow us at @VervericaData

Introducing our new name!
Today, we're excited to share the next phase of our journey - introducing our new name!
ververica.com

7:25 AM - 8 Feb 2019

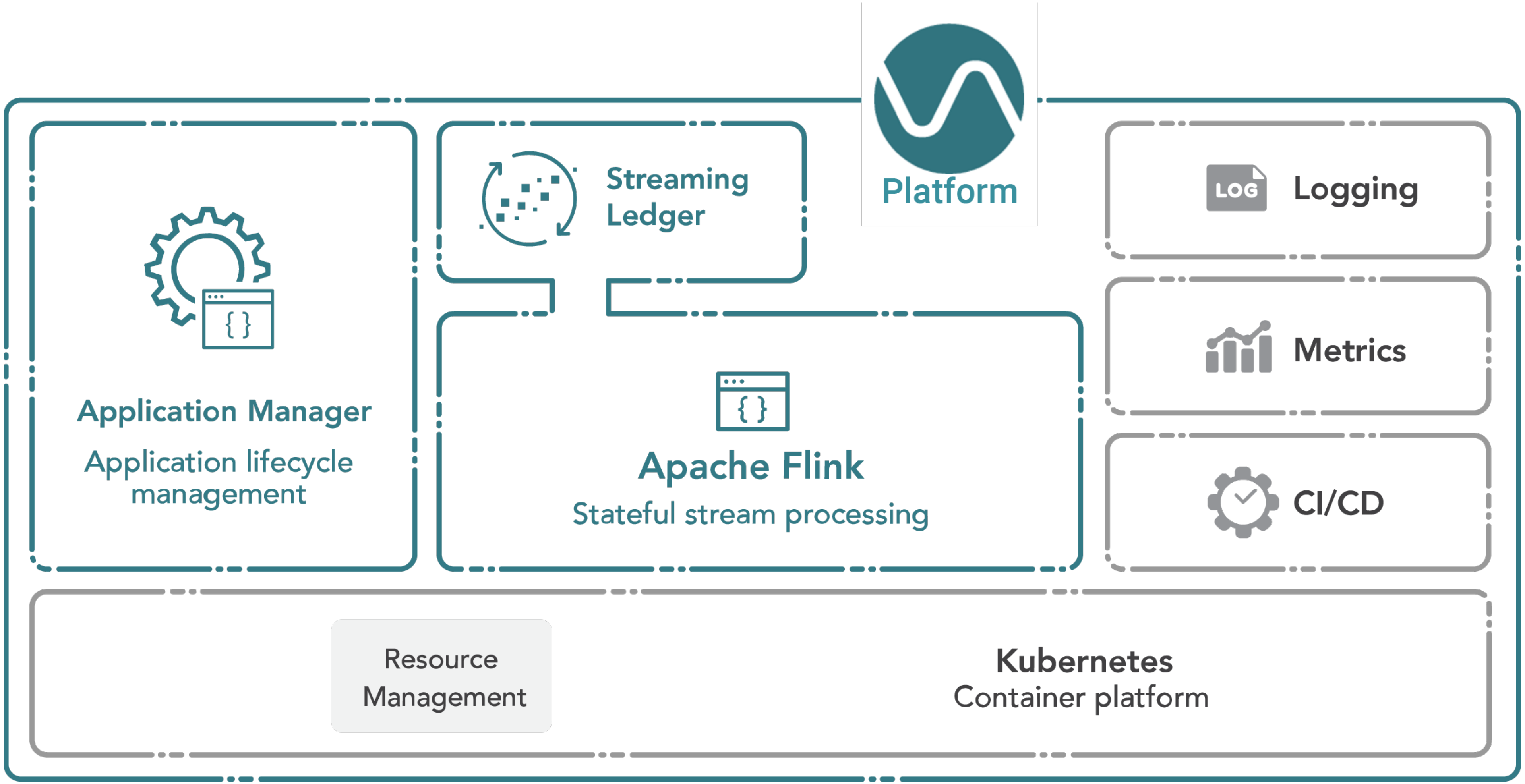
11 Retweets **34** Likes

5 11 34

Kostas Tzoumas @kostas_tzoumas · Feb 8
PS: There is an easter egg in the name. Reply to me if you found it :-)

2 8

dA/ Ververica platform with Streaming Ledger supporting full ACID



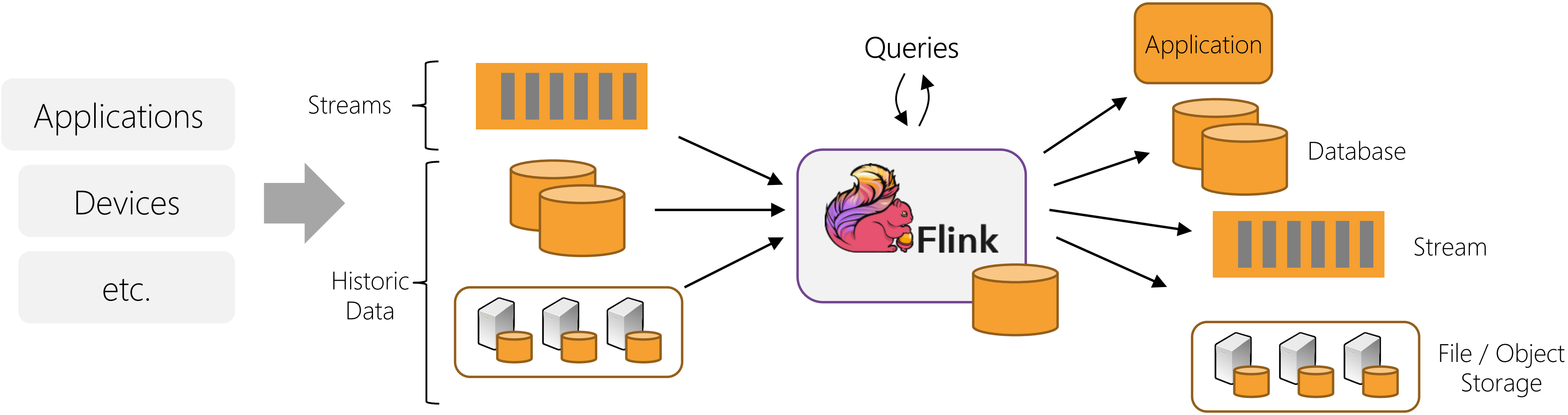
Apache Flink in a Nutshell

Stateful computations over streams

real-time and historic

fast, scalable, fault tolerant, in-memory,

event time, large state, exactly-once

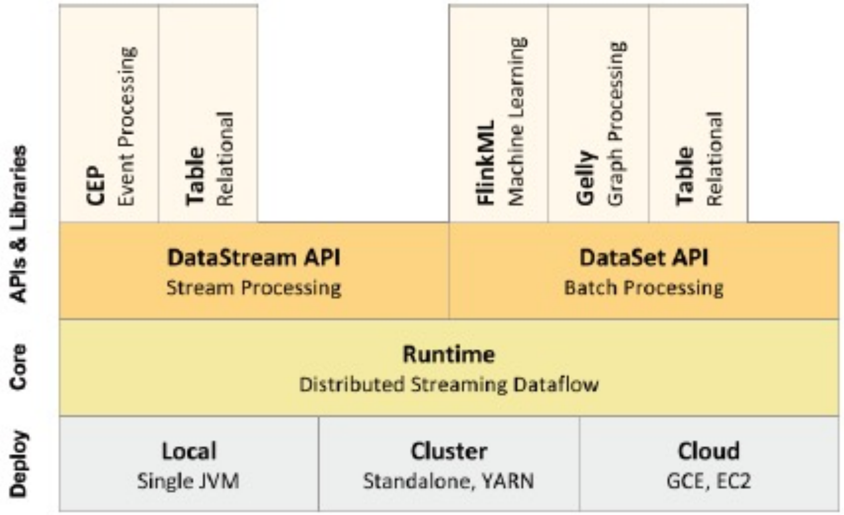


Overview of the Apache Flink Architecture

Apache Flink® is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications.

- Key Features:
  - Bounded and unbounded data
  - Event time semantics
  - Stateful and fault-tolerant
  - Running on thousands of nodes with very good throughput and latency
  - Exactly-once semantics for stateful computations.
  - Flexible windowing based on time, count, or sessions in addition to data-driven windows

- **DataSet** and **DataStream** programming abstractions are the foundation for user programs and higher layers

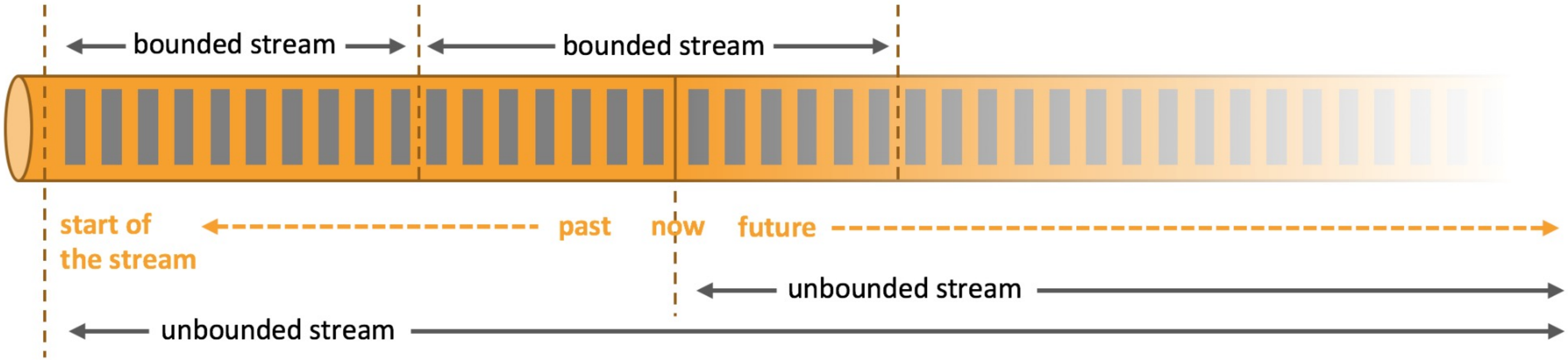


http://flink.apache.org

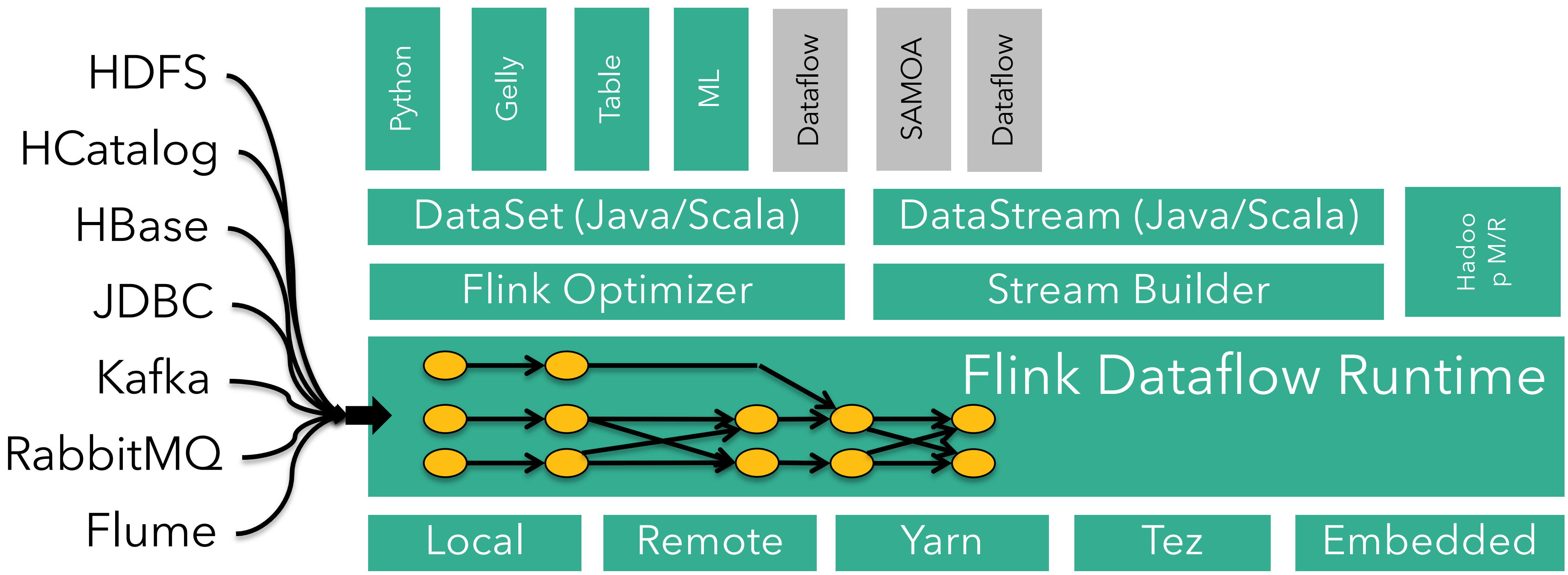
Everything Streams

Apache Flink handles everything as streams internally.

Continuous streaming and applications use "unbounded streams".
Batch processing and finite applications use "bounded streams".



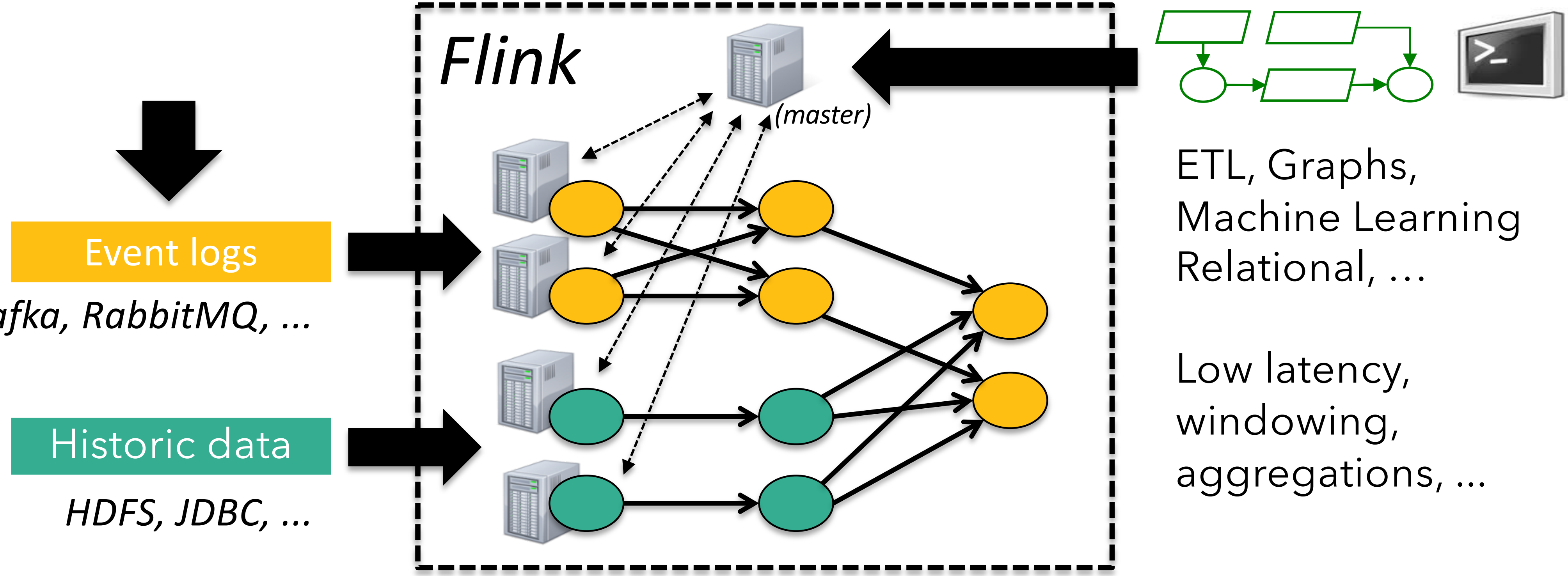
Apache Flink v1.0's Software Stack



Dissecting Flink

What is Apache Flink?

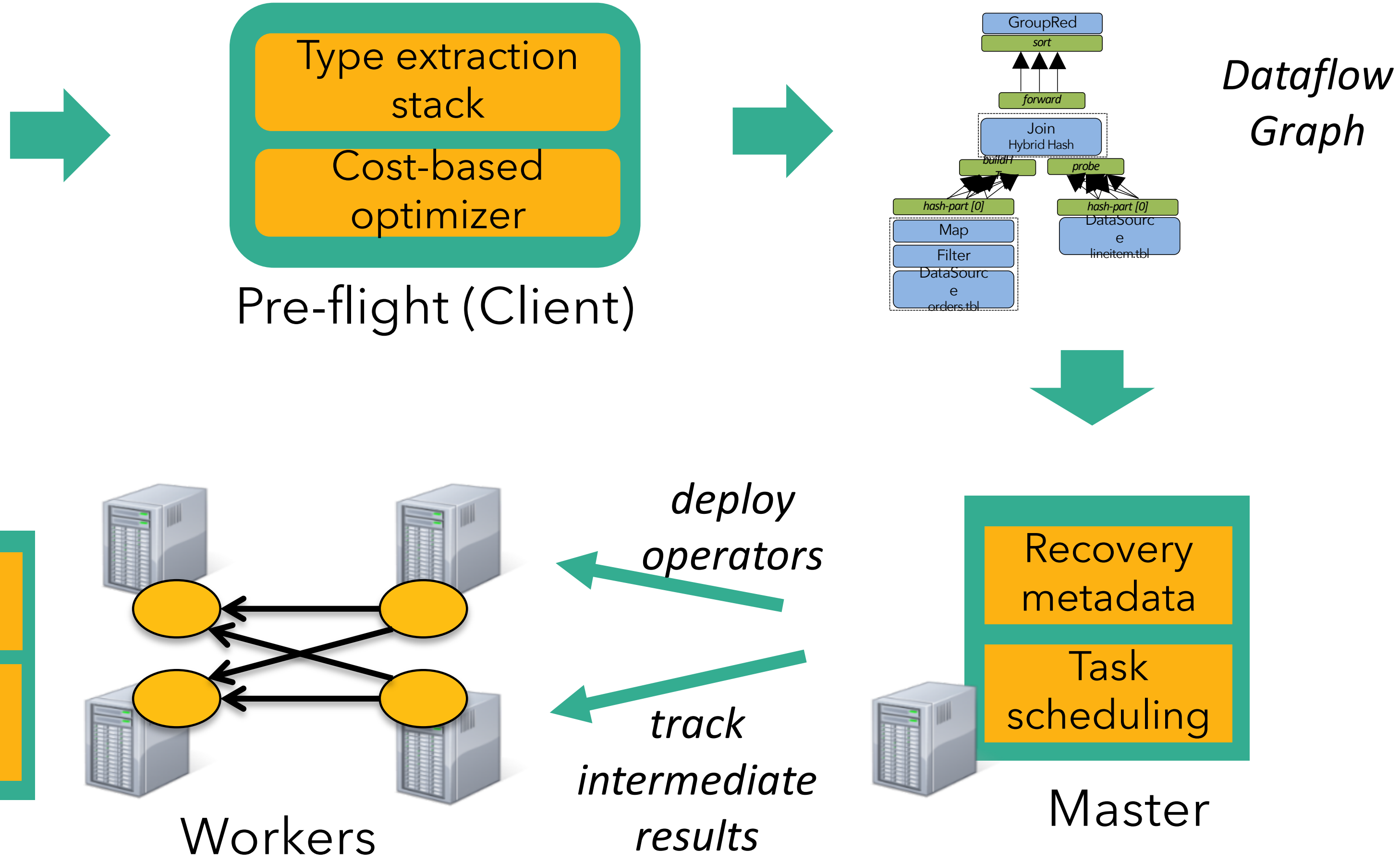
Real-time data streams



Internal Execution Workflow of Flink

```
case class Path (from: Long, to:
Long)
val tc = edges.iterate(10) {
  paths: DataSet[Path] =>
    val next = paths
      .join(edges)
      .where("to")
      .equalTo("from") {
        (path, edge) =>
          Path(path.from, edge.to)
      }
      .union(paths)
      .distinct()
    next
  }
```

Program



Memory manager	Out-of-core algos
Batch & Streaming	State & Checkpoints

P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas:
Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38(4): 28-38 (2015)

Cornerstones of Flink Design

Flexible Data Streaming Engine
- → *Low Latency Stream Proc.*
- → *Highly flexible windowing semantics (i.e. think Beam)*

High-level APIs, beyond key/value pairs
- → *Java, Scala, Python(beta only)*
- → *Relational-style optimizer*

Additional Library Support
- → *Storm Compatibility Library*
- → *Graphs / ML Pipelines*
- → *ML & Streaming ML* ***(Catching up)***

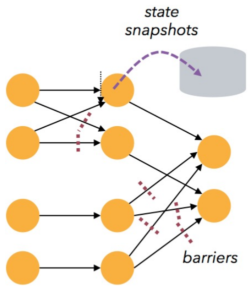
Robust Algorithms on Managed Memory
- → *No OutOfMemory Errors*
- → *Scales to very large JVMs*
- → *Efficient Checkpointing/ Recovery & Saved points Op.*

Pipelined Execution of Batch Programs
- → *Better shuffle performance*
- → *Scales to very large groups*

Native Iterations
- → *Very fast Graph Processing*
- → *Stateful Iterations for ML*

1. Failures and downtime

- Checkpoints & savepoints
- Exactly-once guarantees

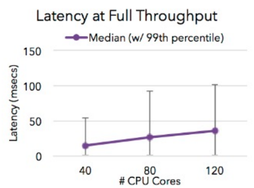


2. Out of order and late data

- Event time support
- Watermarks

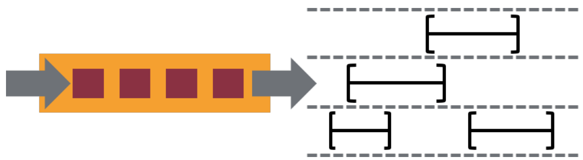
3. Results when you need them

- Low latency
- Triggers



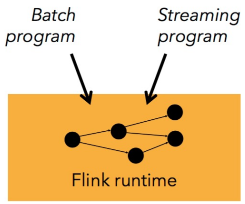
4. Accurate modeling

- True streaming engine
- Sessions and flexible windows



5. Batch + streaming

- One engine
- Dedicated APIs



6. Reprocessing

- High throughput, event time support, and savepoints

```
flink -s <savepoint> <job>
```

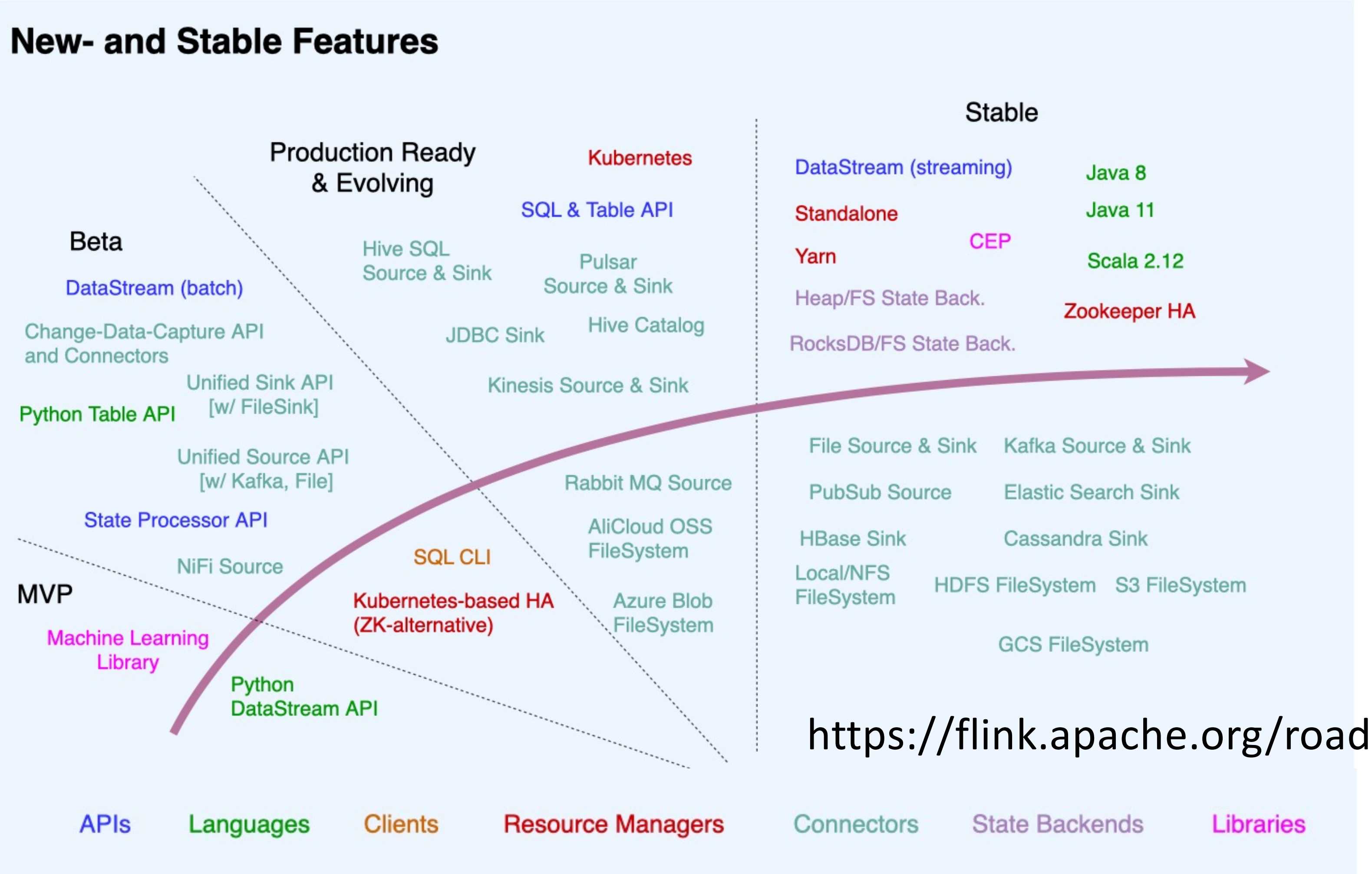
7. Ecosystem

- Rich connector ecosystem and 3rd party packages

8. Community support

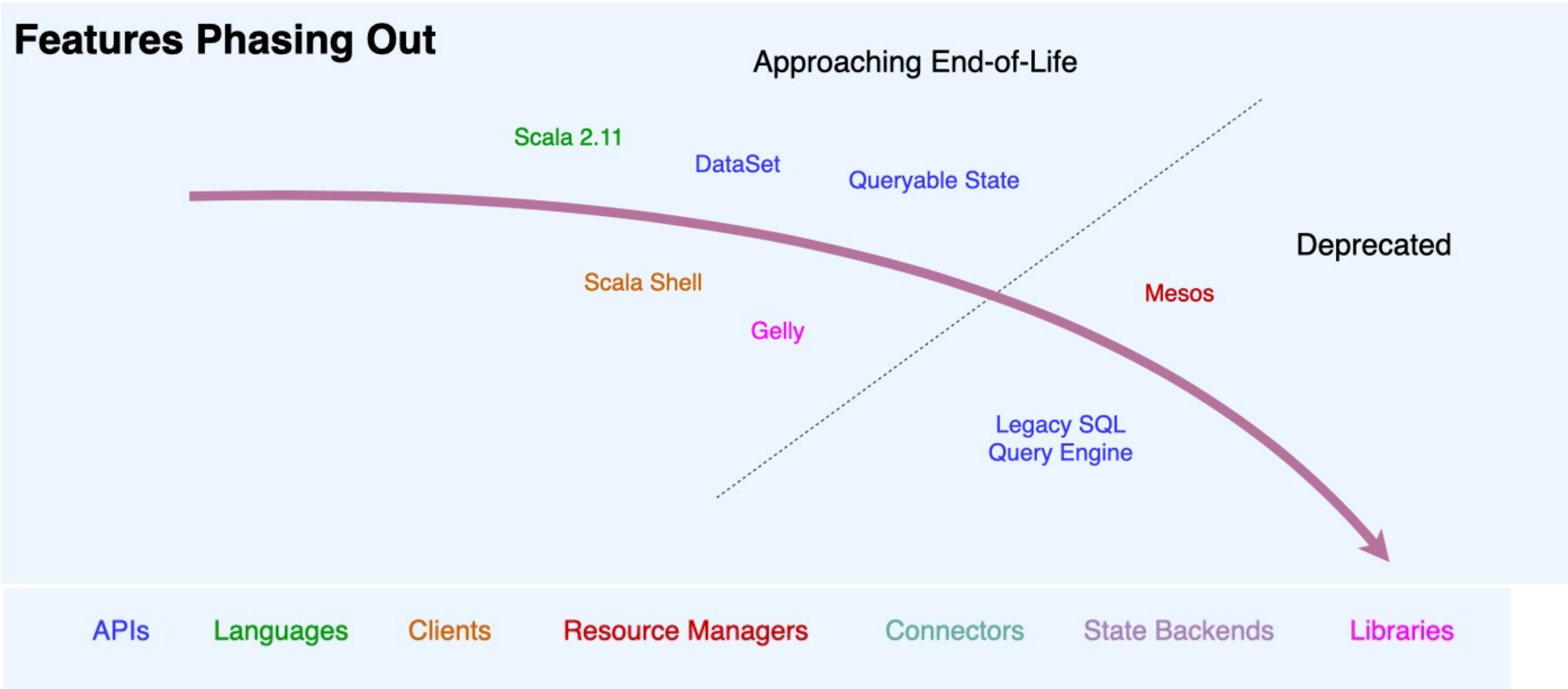
- One of the most active projects with over 200 contributors

Feature Radar of Flink (circa 1Q2021)



https://flink.apache.org/roadmap.html

Feature Radar of Flink (circa 1Q2021)



Using (Programming with) Flink

Layered Abstractions of Flink

Layered abstractions to
navigate simple to complex use cases

```sql
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp)
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

High-level Analytics API	Stream SQL / Tables *(dynamic tables)*
Stream- & Batch Data Processing	DataStream API *(streams, windows)*
Stateful Event-Driven Applications	Process Function *(events, state, time)*

```scala
val stats = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum((a, b) -> a.add(b))
```

```scala
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // work with event and state
  (event, state.value) match { … }

  out.collect(…) // emit events
  state.update(…) // modify state

  // schedule a timer callback
  ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
}
```

Batch / Streaming APIs (Scala)

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment()
val lines: DataSet[String] = env.readTextFile(...)
lines.flatMap {line => line.split(" ")
                           .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

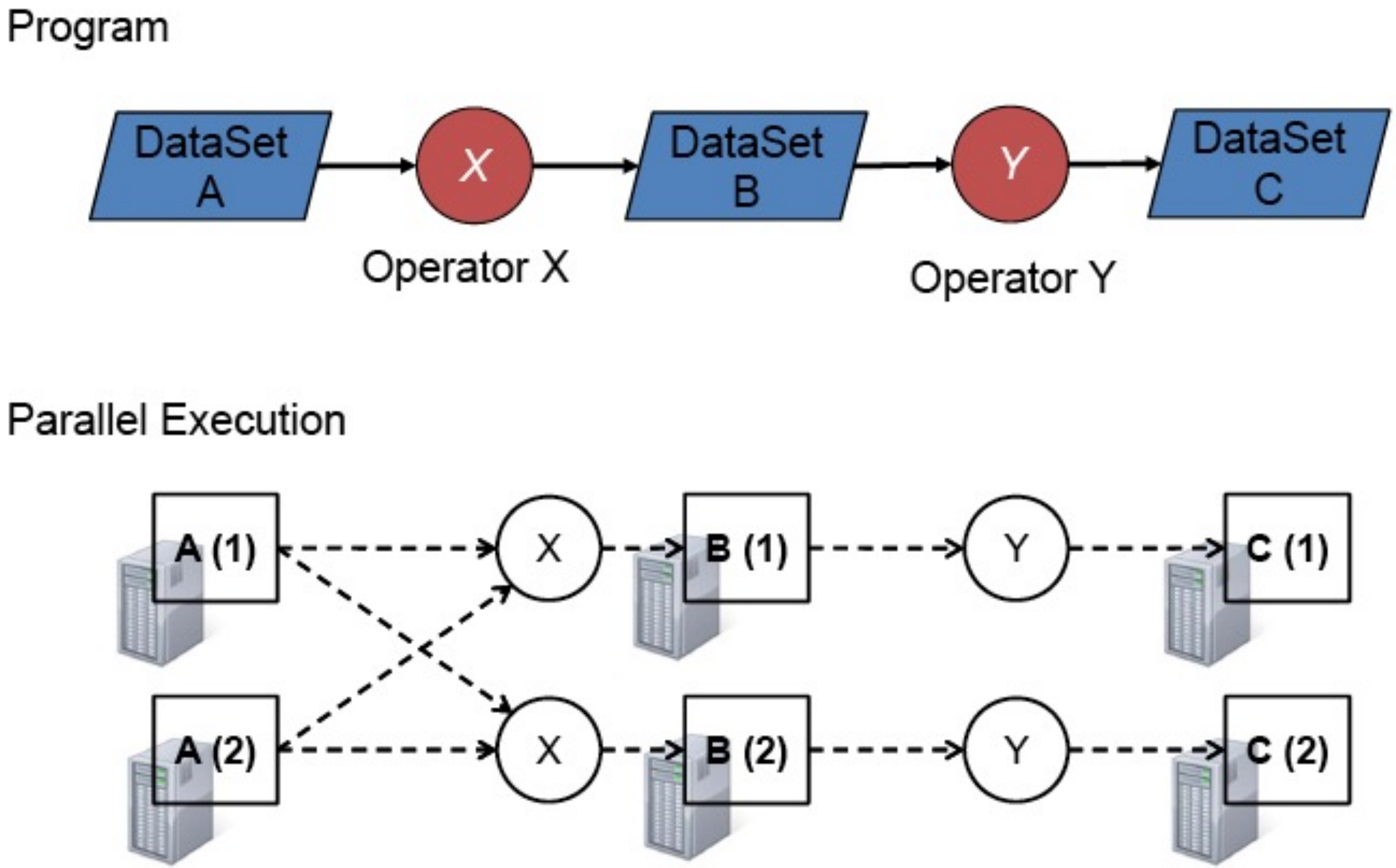
DataStream API (streaming):

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
val lines: DataStream[String] = env.fromSocketStream(...)
lines.flatMap {line => line.split(" ")
                           .map(word => Word(word,1))}
    .keyBy("word")
    .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
    .sum("frequency")
    .print()
```

Batch & Streaming

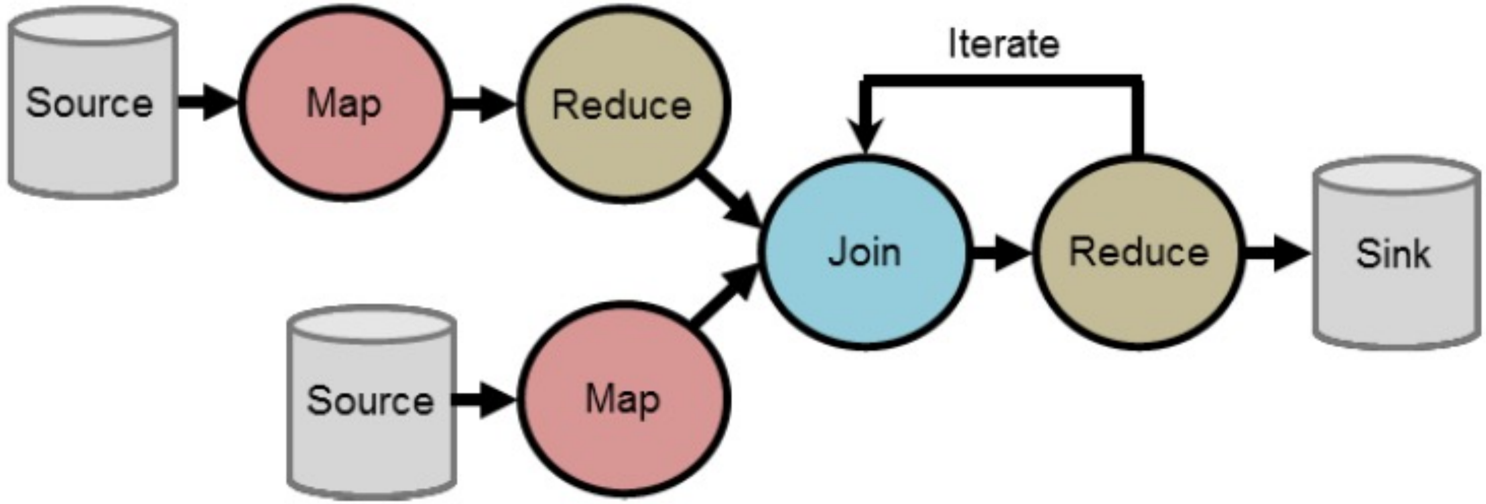
	Streaming	Batch
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

Data sets and Operators

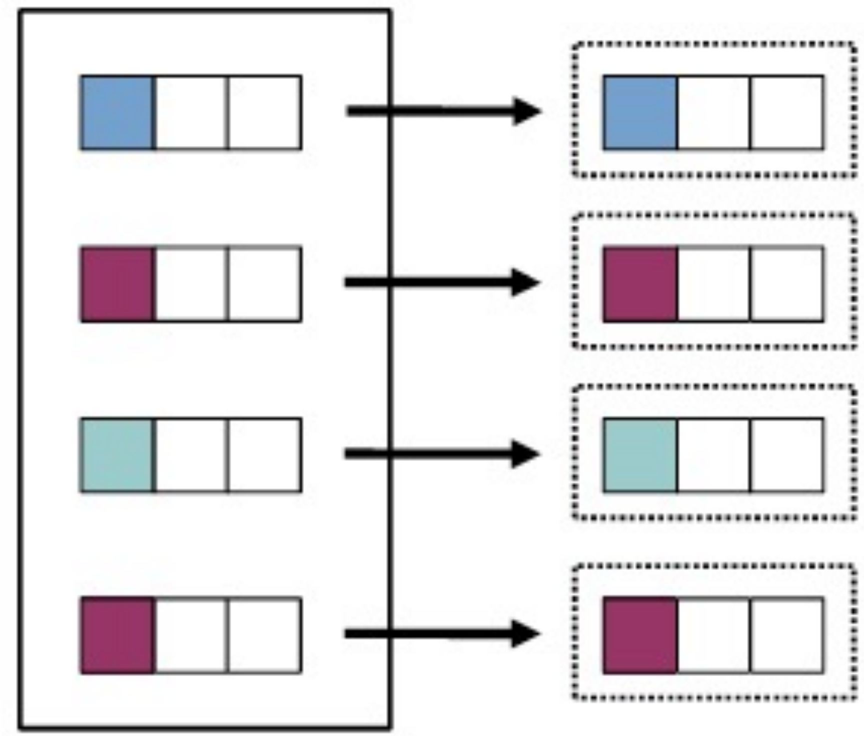


Flink's set of Operators

Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators, …

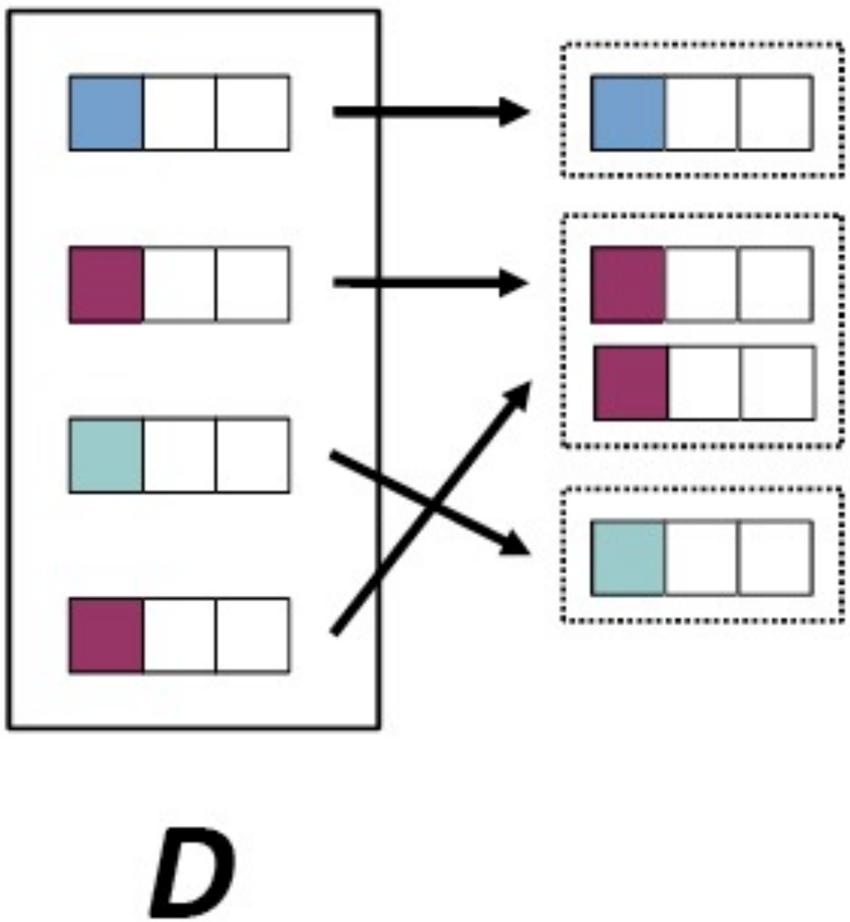


Base-Operator: Map

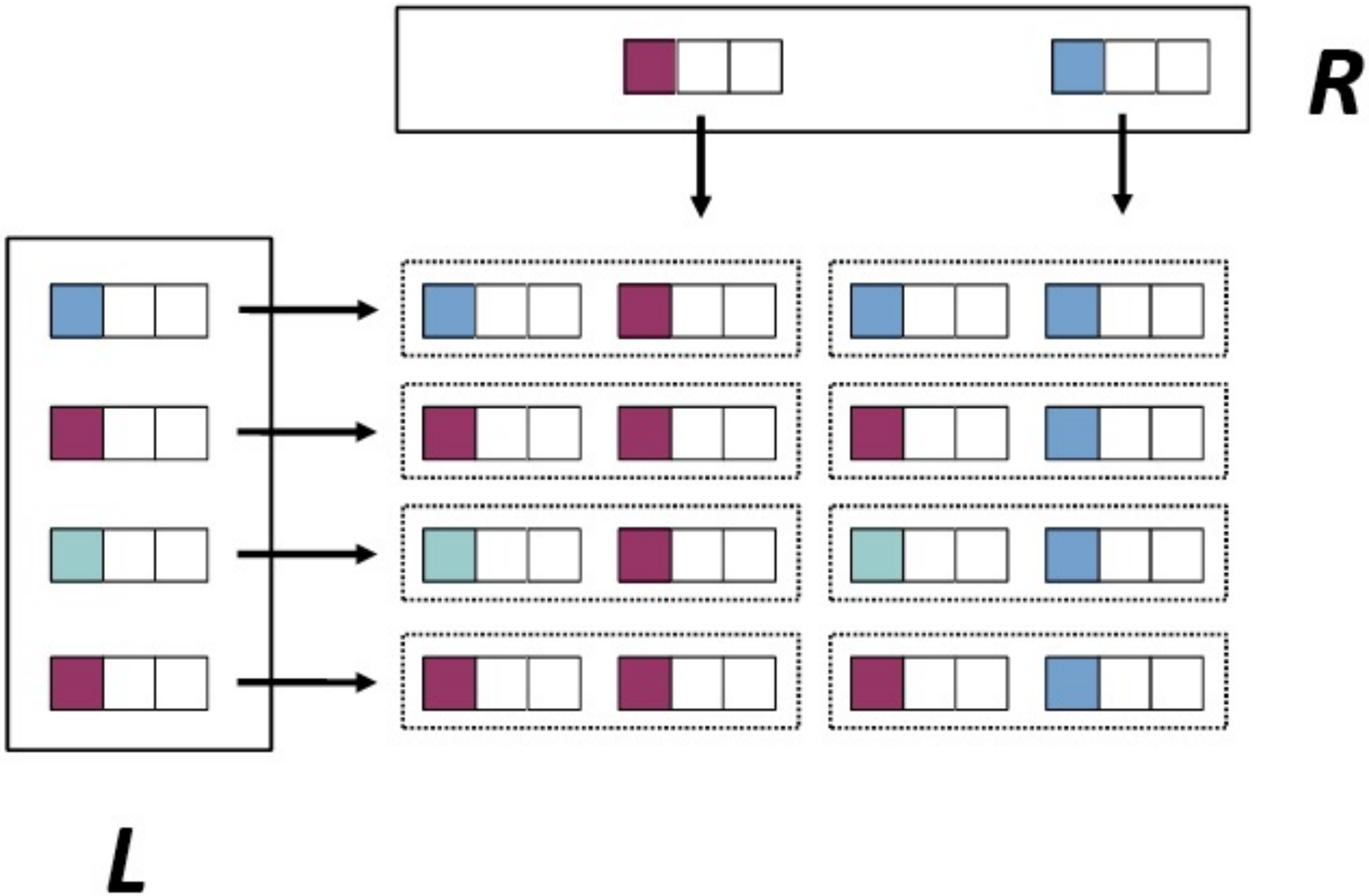


D

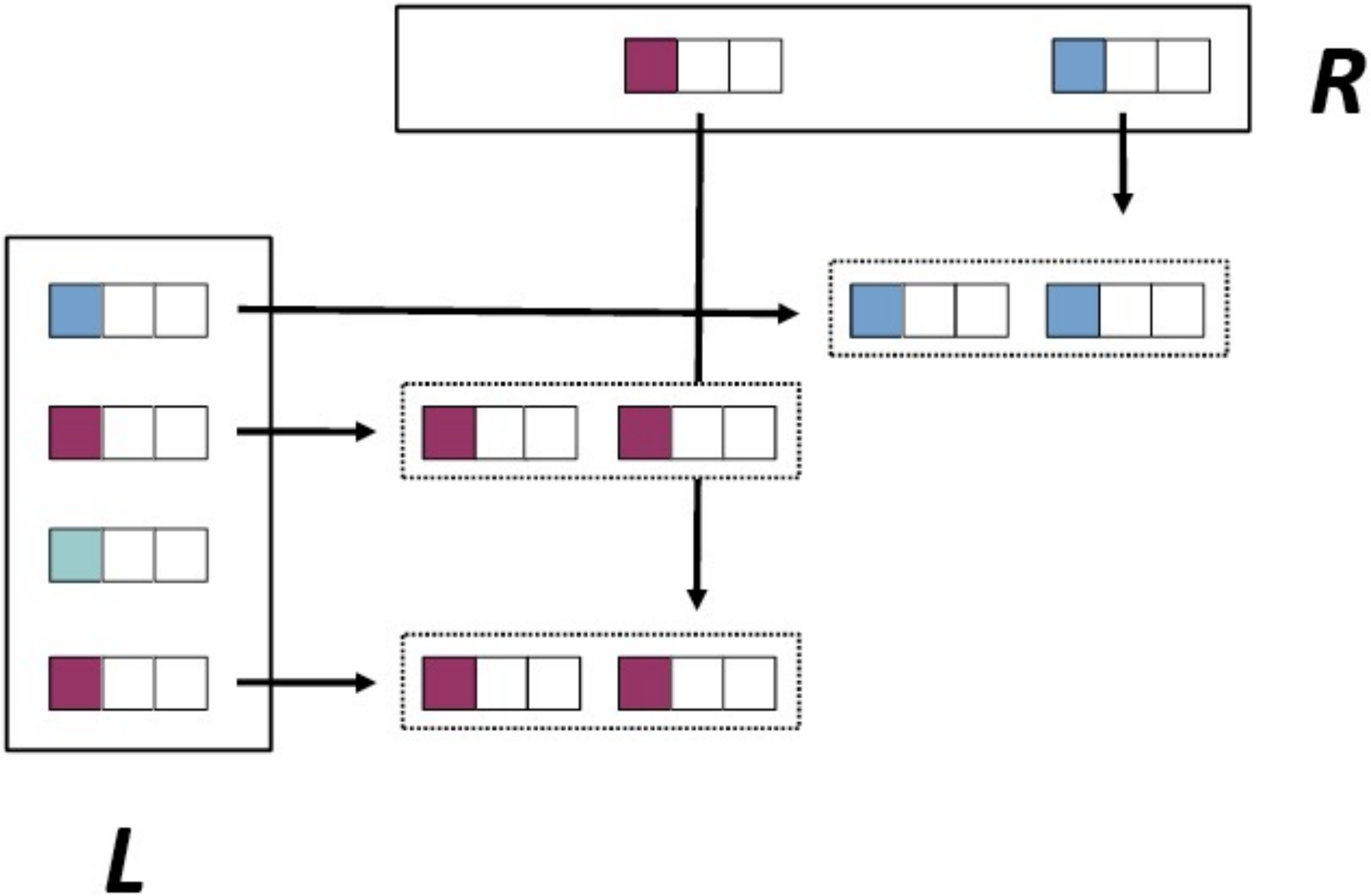
Base-Operator: Reduce



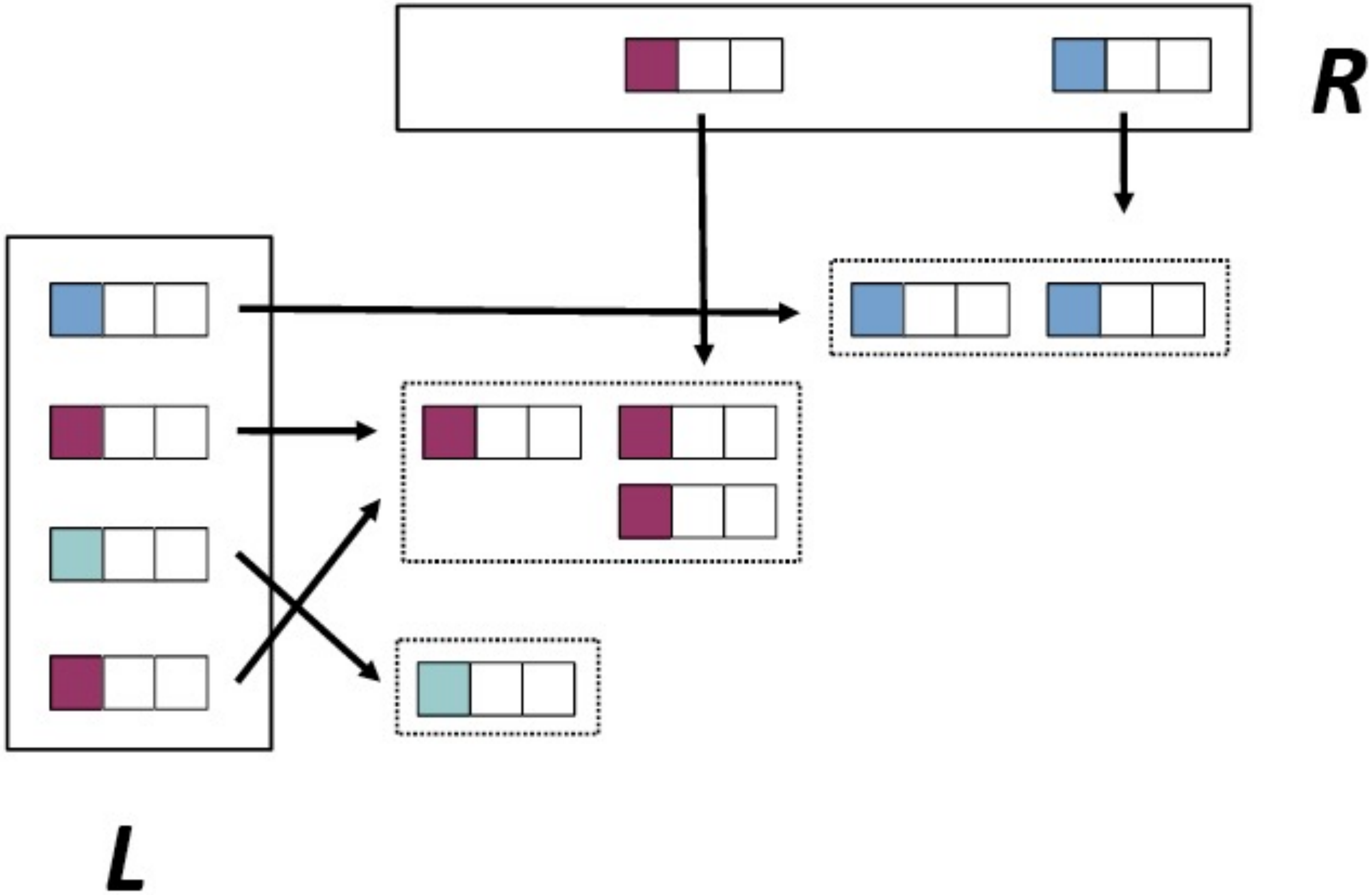
Base-Operator: Cross



Base-Operator: Join



Base-Operator: CoGroup

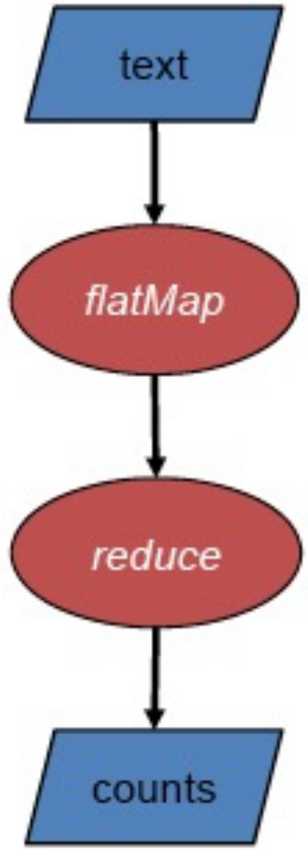


DataSet and transformations (Java)


```
ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();
DataSet<String> input = env.readTextFile(input);

DataSet<String> first = input
    .filter (str -> str.contains(“Apache Flink“));
DataSet<String> second = first
    .filter (str -> str.length() > 40);

second.print();
env.execute();
```

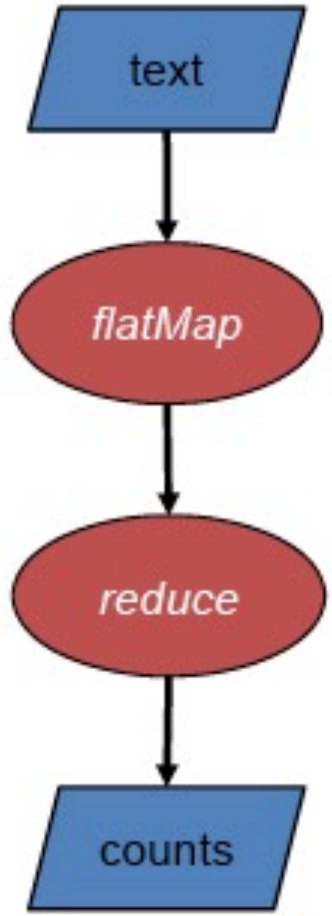
WordCount in Java (with DataSet)

```java
ExecutionEnvironment env =
  ExecutionEnvironment.getExecutionEnvironment();

DataSet<String> text = readTextFile (input);

DataSet<Tuple2<String, Integer>> counts= text
   .map (l -> l.split("\\W+"))
   .flatMap ((String[] tokens,
               Collector<Tuple2<String, Integer>> out) -> {
      Arrays.stream(tokens)
      .filter(t -> t.length() > 0)
      .forEach(t -> out.collect(new Tuple2<>(t, 1)));
    })
   .groupBy(0)
   .sum(1);

env.execute("Word Count Example");
```



WordCount in Scala (with DataSet)

```
val env = ExecutionEnvironment
  .getExecutionEnvironment

val input = env.readTextFile(textInput)

val counts = text
  .flatMap { line => line.split("\\W+") }
  .filter { term => term.nonEmpty }
  .map { term => (term, 1) }
  .groupBy(0)
  .sum(1)

env.execute()
```



Another Example: Transitive Closure (Java)

```
IterativeDataSet<Tuple2<Long,Long>> paths = edges.iterate (10);

DataSet<Tuple2<Long,Long>> nextPaths = paths
   .join(edges).where(1).equalTo(0)
   .with((left, right) -> return new Tuple2<Long, Long>(left.f0, right.f1);)
   .union(paths)
   .distinct();

DataSet<Tuple2<Long, Long>> tc = paths.closeWith(nextPaths);
```

Transitive Closure (Scala)

Transitive Closure

```
case class Path (from: Long, to: Long)
val tc = edges.iterate(10) { paths: DataSet[Path] =>
  val next = paths
    .join(edges).where("to").equalTo("from") {
      (path, edge) => Path(path.from, edge.to)
    }
    .union(paths).distinct()
  next
}
```

More Details on the Flink API

DataSet

- Central notion of the batch-based programming API
- Files and other data sources are read into DataSets
  - `DataSet<String> text = env.readTextFile(…)`
- Transformations on DataSets produce DataSets
  - `DataSet<String> first = text.map(…)`
- DataSets are printed to files or on stdout
  - `first.writeAsCsv(…)`
- Execution is triggered with **`env.execute()`**

Data Types

- Basic Java Types
  - String, Long, Integer, Boolean, ...
  - Arrays
- Composite Types
  - Tuple
  - PoJo (Java Objects)
  - Custom type

Data Types - Tuples

- Bean-style Java classes & field names
- Tuples and position addressing
- Any data type with key selector function
- Easy, lightweight and generic way of encapsulating data in Flink
  - Tuple1 upto Tuple25

Example:

```
Tuple3<String, String, Integer> person =
            new Tuple3<>("Max", "Magmum", 42) ;
      // zero-based index !
      String firstName = person.f0 ;
      String secondName = person.f1 ;
      Integer age = person.f2 ;
```

Beyond Key/Value Pairs

```
DataSet<Page> pages = ...;
DataSet<Impression> impressions = ...;

DataSet<Impression> aggregated =
 impressions
  .groupBy("url")
  .sum("count");

pages.join(impressions).where("url").equalTo("url")


// custom data types

class Impression {
    public String url;
    public long count;
}
```

```
class Page {
    public String url;
    public String topic;
}
```

Data types and grouping

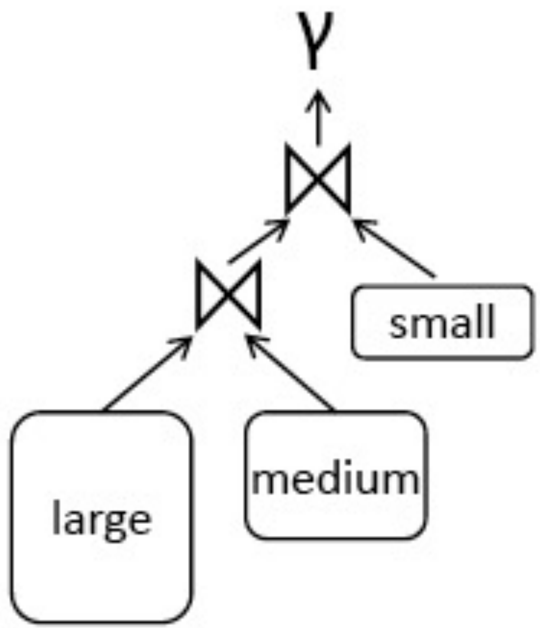
```
public static class Access {
  public int userId;
  public String url;
  ...
}
```

```
public static class User {
  public int userId;
  public int region;
  public Date customerSince;
  ...
}
```

```
DataSet<Tuple2<Access,User>> campaign = access.join(users)
  .where(“userId“).equalTo(“userId“)

DataSet<Tuple3<Integer,String,String> someLog;
someLog.groupBy(0,1).reduceGroup(...);
```

Long Operator Pipelines

```
DataSet<Tuple...> large = env.readCsv(...);
DataSet<Tuple...> medium = env.readCsv(...);
DataSet<Tuple...> small = env.readCsv(...);

DataSet<Tuple...> joined1 =
        large.join(medium)
             .where(3).equals(1)
             .with(new JoinFunction() { ... });

DataSet<Tuple...> joined2 =
         small.join(joined1)
              .where(0).equals(2)
              .with(new JoinFunction() { ... });

DataSet<Tuple...> result = joined2.groupBy(3)
                                  .max(2);
```



Available transformations

- map
- flatMap
- filter
- reduce
- reduceGroup
- join
- coGroup
- aggregate
- cross
- project
- distinct
- union
- iterate
- iterateDelta
- repartition
- …

Transformations: Map

```
DataSet<Integer> integers = env.fromElements(1, 2, 3, 4);

// Regular Map - Takes one element and produces one element
DataSet<Integer> doubleIntegers =
     integers.map(new MapFunction<Integer, Integer>() {
     @Override
         public Integer map(Integer value) {
            return value * 2;
         }
     });

doubleIntegers.print();
> 2, 4, 6, 8


// Flat Map - Takes one element and produces zero, one, or more elements.
DataSet<Integer> doubleIntegers2 =

     integers.flatMap(new FlatMapFunction<Integer, Integer>() {
          @Override
          public void flatMap(Integer value, Collector<Integer> out) {
               out.collect(value * 2);
          }
     });

doubleIntegers2.print();
> 2, 4, 6, 8
```

Transformations: Filter

```
// The DataSet
DataSet<Integer> integers = env.fromElements(1, 2, 3, 4);

DataSet<Integer> filtered =

    integers.filter(new FilterFunction<Integer>() {
         @Override
         public boolean filter(Integer value) {
              return value != 3;
         }
    });

integers.print();
> 1, 2, 4
```

Transformations: Group and Reduce

- DataSets can be split into groups
- Groups are defined using a common key

Name	Age
Stephan	18
Fabian	23
Julia	27
Romeo	27
Anna	18

```
// (name, age) of employees
DataSet<Tuple2<String, Integer>> employees = …

// group by second field (age)
DataSet<Integer, Integer> grouped = employees.groupBy(1)
  // return a list of age groups with its counts
  .reduceGroup(new CountSameAge());
```

AgeGroup	Count
18	2
23	1
27	2

Transformations: GroupReduce

```
public static class CountSameAge implements GroupReduceFunction
<Tuple2<String, Integer>, Tuple2<Integer, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<Integer, Integer>> out) {

        Integer ageGroup = 0;
        Integer countsInGroup = 0;

        for (Tuple2<String, Integer> person : values) {
            ageGroup = person.f1;
            countsInGroup++;
        }

        out.collect(new Tuple2<Integer, Integer>
                                               (ageGroup, countsInGroup));
    }
}
```

Transformations: Joining 2 DataSets

Authors		
Id	Name	email
1	Fabian	fabian@..
2	Julia	julia@...
3	Max	max@...
4	Romeo	romeo@.

Posts		
Title	Content	Author id
...	...	2
..	..	4
..	..	4
..	..	1
..	..	2

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

DataSet<Tuple2<
           Tuple3<Integer, String, String>,
           Tuple3<String, String, Integer>
       >> archive = authors.join(posts).where(0).equalTo(2);
```

Transformations: Joining 2 DataSets

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

DataSet<Tuple2<
            Tuple3<Integer, String, String>,
            Tuple3<String, String, Integer>
        >> archive = authors.join(posts).where(0).equalTo(2);
```

Archive					
Id	Name	email	Title	Content	Author id
1	Fabian	fabian@..	..	..	1
2	Julia	julia@...	..	..	2
2	Julia	julia@...	..	..	2
3	Romeo	romeo@...	..	..	4
4	Romeo	romeo@.	..	..	4

Transformations: Join with join function

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

// (title, author name)
DataSet<Tuple2<String, String>> archive =
    authors.join(posts).where(0).equalTo(2)
    .with(new PostsByUser());

public static class PostsByUser implements
  JoinFunction<Tuple3<Integer, String, String>,
               Tuple3<String, String, Integer>,
               Tuple2<String, String>> {
    @Override
    public Tuple2<String, String> join(
        Tuple3<Integer, String, String> left,
        Tuple3<String, String, Integer> right) {
            return new Tuple2<String, String>(left.f1, right.f0);
    }
}
```

Archive	
Name	Title
Fabian	..
Julia	..
Julia	..
Romeo	..
Romeo	..

Data Sources

Batch API

- Files
  - HDFS, Local file system, MapR file system
  - Text, Csv, Avro, Hadoop input formats
- JDBC
- HBase
- Collections

Stream API

- Files
- Socket streams
- Kafka
- RabbitMQ
- Flume
- Collections
- Implement your own
  - SourceFunction.collect

Data Sources

Text

- `readTextFile(“/path/to/file”)`

CSV

- `readCsvFile(“/path/to/file”)`

Collection

- `fromCollection(collection)`
- `fromElements(1,2,3,4,5)`

Data Sources: Collections

```
ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();

// read from elements
DataSet<String> names = env.fromElements("Some", "Example",
"Strings");

// read from Java collection
List<String> list = new ArrayList<String>();
list.add("Some");
list.add("Example");
list.add("Strings");

DataSet<String> names = env.fromCollection(list);
```

Data Sources: File-based

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();


// read text file from local or distributed file system
DataSet<String> localLines =
    env.readTextFile("/path/to/my/textfile");


// read a CSV file with three fields
DataSet<Tuple3<Integer, String, Double>> csvInput =
                 env.readCsvFile("/the/CSV/file")
                 .types(Integer.class, String.class, Double.class);


// read a CSV file with five fields, taking only two of them
DataSet<Tuple2<String, Double>> csvInput =
                 env.readCsvFile("/the/CSV/file")
                 // take the first and the fourth field
                 .includeFields("10010")
                 .types(String.class, Double.class);
```

Data Sinks

Text

- `writeAsText("/path/to/file")`
- `writeAsFormattedText("/path/to/file", formatFunction)`

CSV

- `writeAsCsv("/path/to/file")`

Return data to the Client

- `Print()`
- `Collect()`
- `Count()`

Data Sinks (lazy)

- Lazily executed when `env.execute()` is called

```
DataSet<…> result;

// write DataSet to a file on the local file system
result.writeAsText("/path/to/file");

// write DataSet to a file and overwrite the file if it exists
result.writeAsText("/path/to/file",FileSystem.WriteMode.OVERWRITE);

// tuples as lines with pipe as the separator "a|b|c"
result.writeAsCsv("/path/to/file", "\n", "|");

// this wites values as strings using a user-defined TextFormatter
object
result.writeAsFormattedText("/path/to/file",
        new TextFormatter<Tuple2<Integer, Integer>>() {
            public String format (Tuple2<Integer, Integer> value) {
                return value.f1 + " - " + value.f0;
            }
        });
```

Data Sinks (eager)

- Eagerly executed

```
DataSet<Tuple2<String, Integer> result;


// print
result.print();

// count
int numberOfElements = result.count();

// collect
List<Tuple2<String, Integer> materializedResults = result.collect();
```

More Details: WordCount's main( ) in Java

```java
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
       ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execution Environment

```java
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data Sources

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data Types

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Transformations

```java
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

User Functions

```java
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

DataSinks

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execute !

```java
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

WordCount: Map

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
          if (token.length() > 0) {
              out.collect(
                 new Tuple2<String, Integer>(token, 1));
          }
      }
  }
}
```

WordCount: Map: Interface

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Map: Types

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Map: Collector

```java
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                    new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Reduce

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
        int count = 0;
        String word = null;
        for (Tuple2<String, Integer> tuple : values) {
            word = tuple.f0;
            count++;
        }
        out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Interface

```java
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
        int count = 0;
        String word = null;
        for (Tuple2<String, Integer> tuple : values) {
            word = tuple.f0;
            count++;
        }
        out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Types

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
        int count = 0;
        String word = null;
        for (Tuple2<String, Integer> tuple : values) {
            word = tuple.f0;
            count++;
        }
        out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Collector

```java
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
        int count = 0;
        String word = null;
        for (Tuple2<String, Integer> tuple : values) {
            word = tuple.f0;
            count++;
        }
        out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

Real-time Streaming with Flink

Flink Real-time Streaming overview

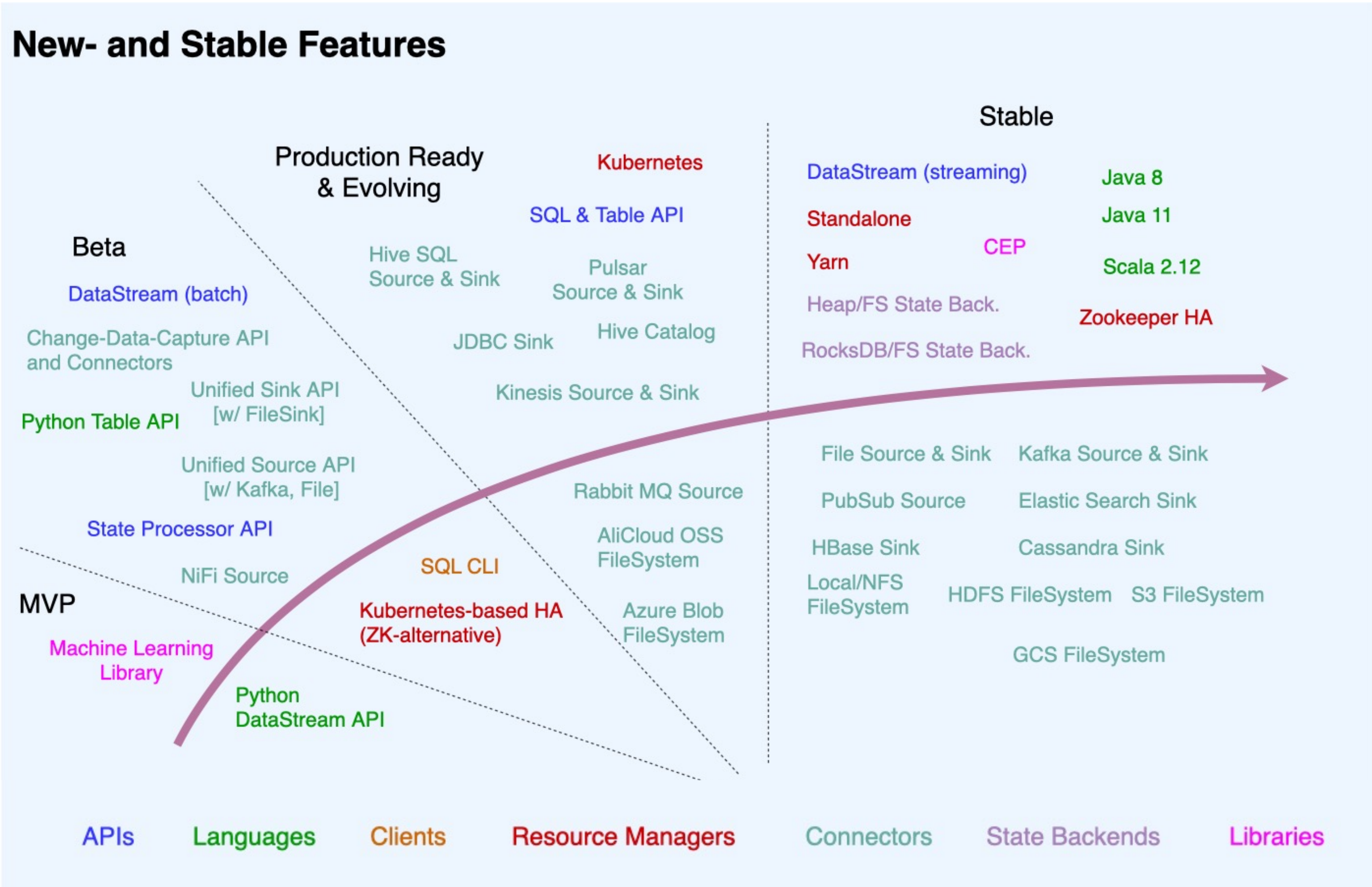
- Historically, Flink first supported Batch (via PACT etc) and Streaming was added later on.

- Streaming and Batch use same code paths in runtime

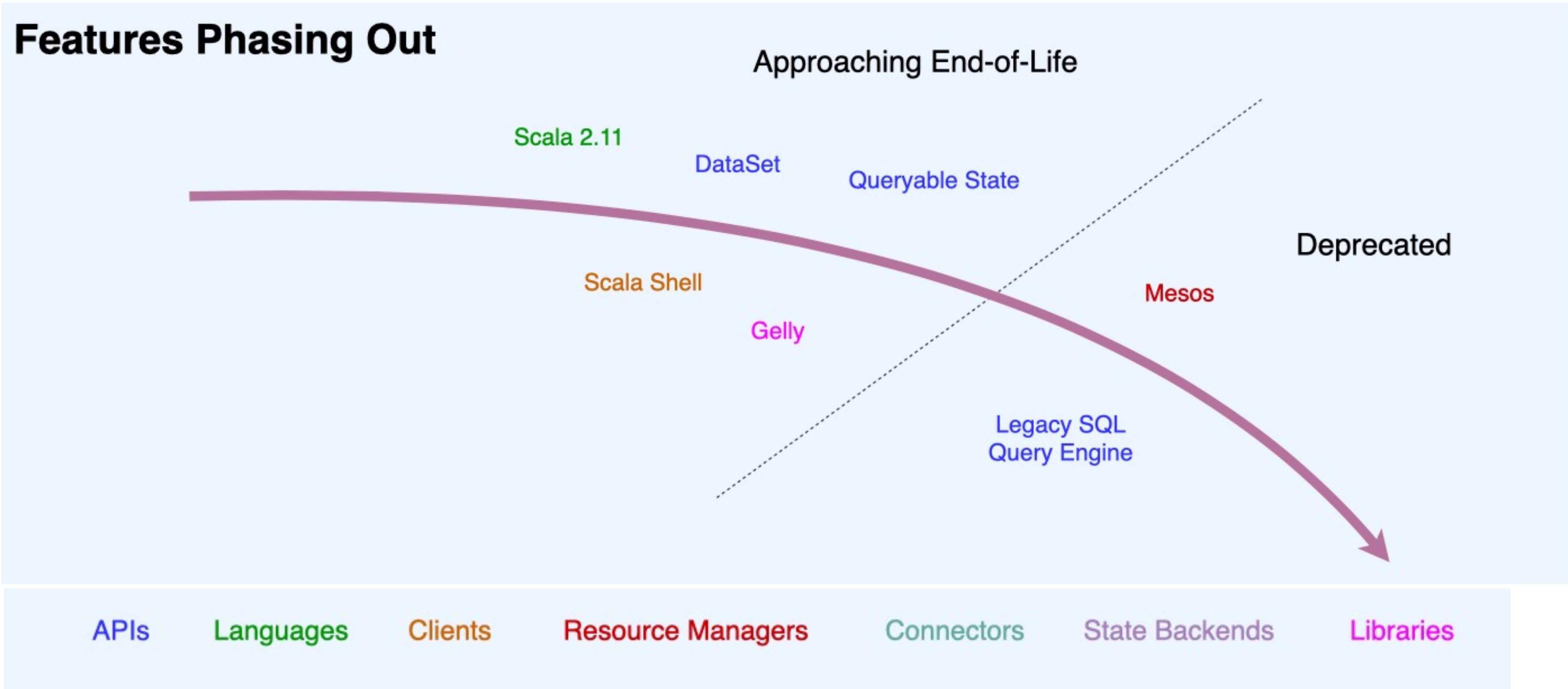
- Differences
  - Streaming does not use Flink's memory management
  - Streaming uses its own compiler/ optimizer

- Alibaba has been working on unifying the Batch and Streaming APIs of Flink
  - The plan is for Flink to just use a single Unified Streaming API for EVERYTHING (but still work in progress) !
  - https://files.alicdn.com/tpsservice/8510c65ffa1fde57274595c5bb009347.pdf

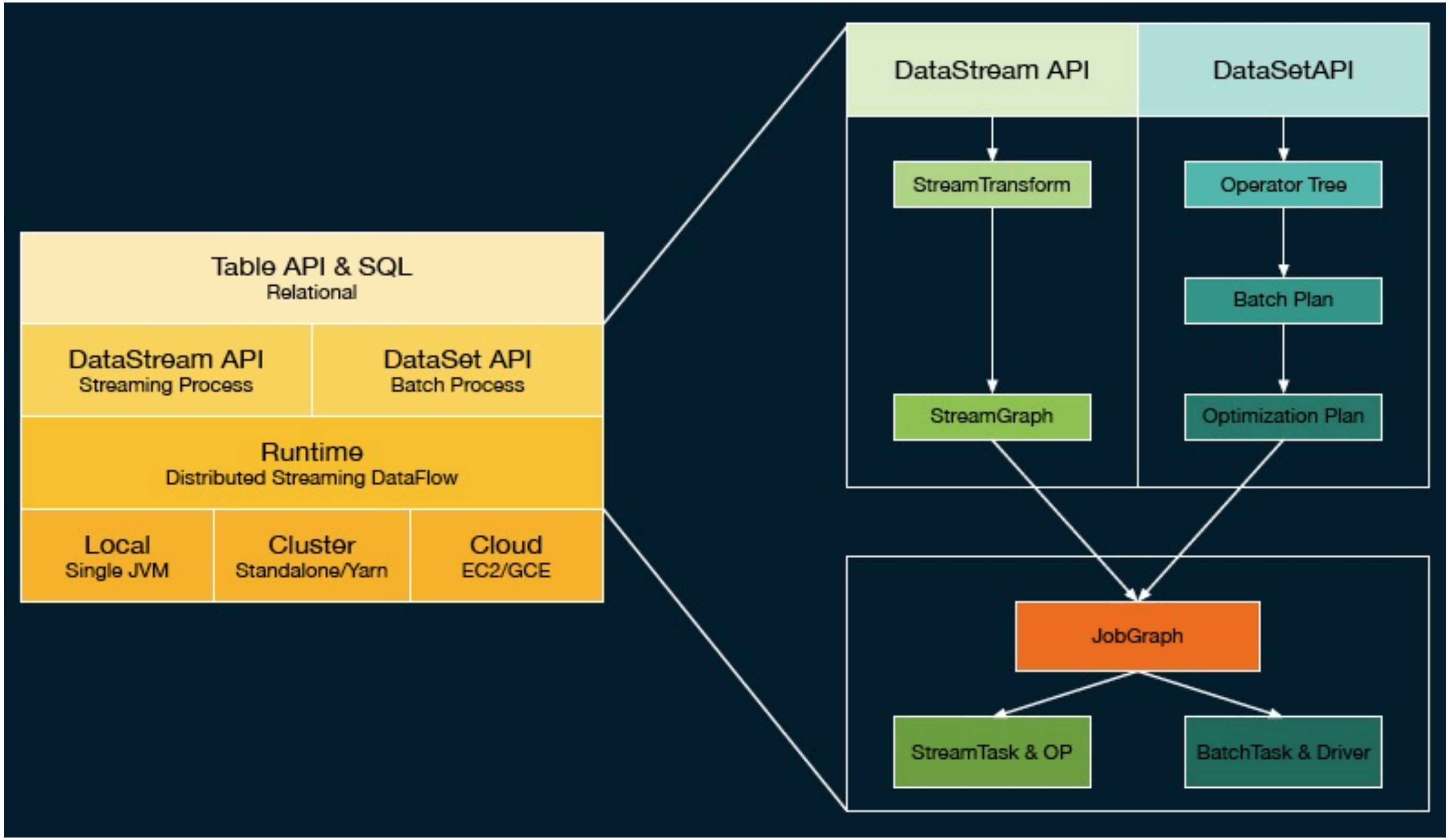
Feature Radar of Flink (circa 1Q2021)



Feature Radar of Flink (circa 1Q2021)



Current Flink API Stack (circa 1Q2019)



Current real-time stream processing

DataStream instead of **DataSet**

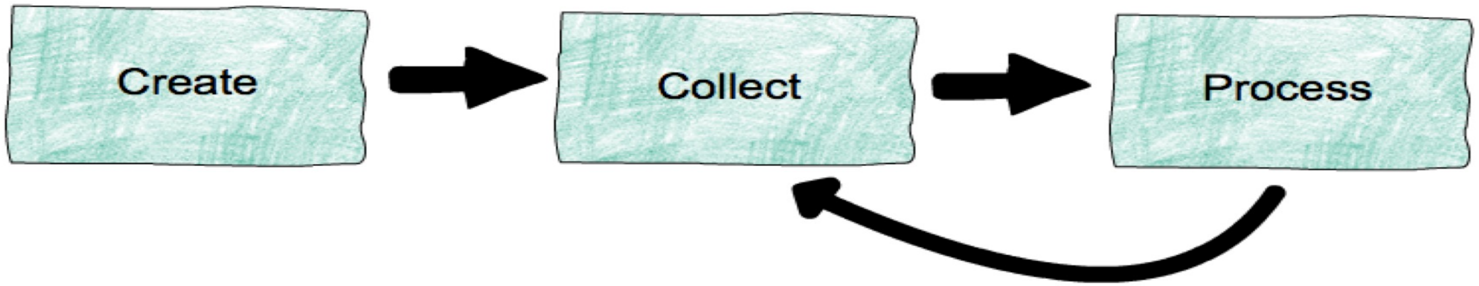
StreamExecutionEnvironment instead of **ExecutionEnvironment**

```
StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> tweets = env.socketTextStream(host,port);

DataStream<Tuple2<String,Integer>> filteredTweets = tweets
  .flatMap(new SelectLanguageAndTokenize())
  .partition(0)
  .map(s -> new Tuple2<String,Integer>(s, 1))
  .groupBy(0).sum(1)
  .flatMap(new SelectMaxOccurence());

tweets.print();
env.execute();
```

Streaming operators

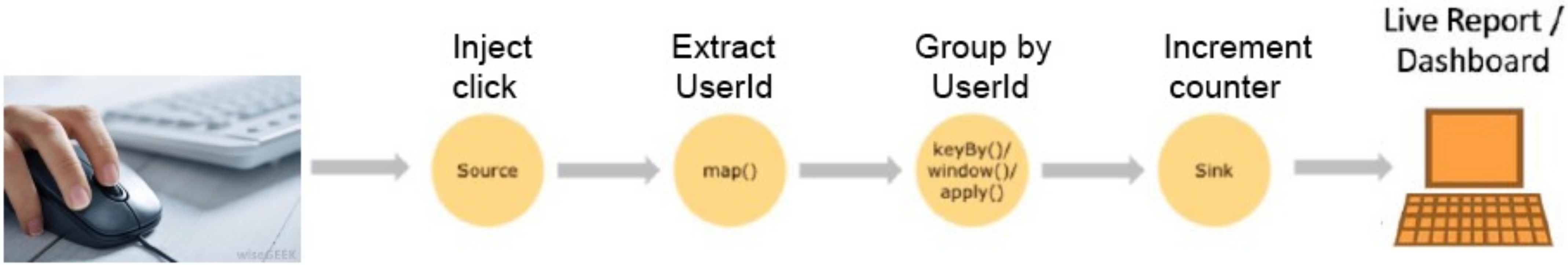
- Most DataSet operators can be used
  - map, filter, flatMap, reduce, reduceGroup, join, cross, coGroup, iterate, project, grouping, partitioning, aggregations, union (merge), …
- DataStream-specific operators (snip)
  - CoMap, CoReduce, etc: share state between streams
  - Temporal binary ops: join, cross, …
  - Windows: policy-based flexible windowing
    - Time, Count, Delta

Life of data streams

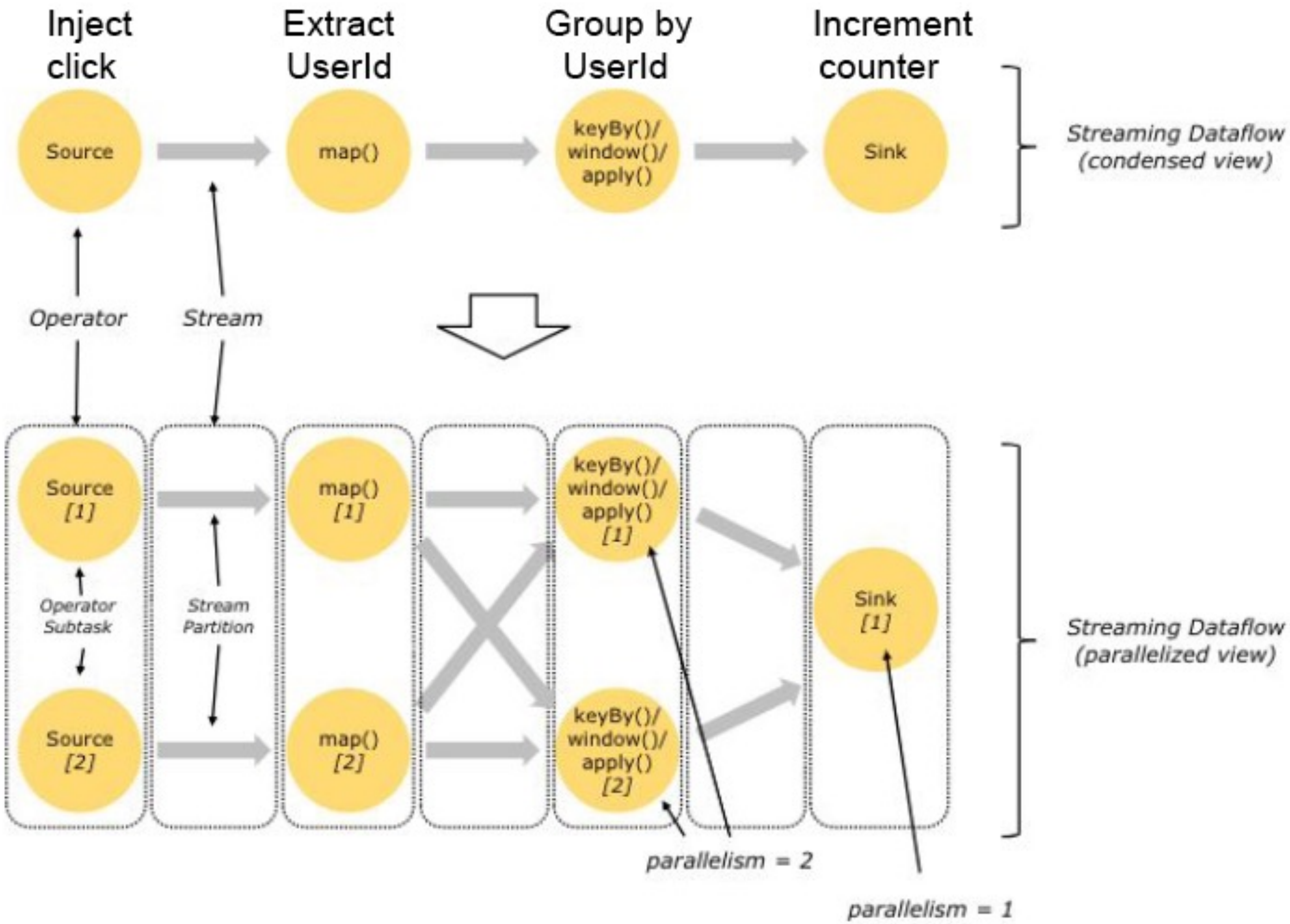


- Create: create streams from event sources (machines, databases, logs, sensors, …)
- Collect: collect and make streams available for consumption (e.g., Apache Kafka)
- Process: process streams, possibly generating derived streams (e.g., Apache Flink)

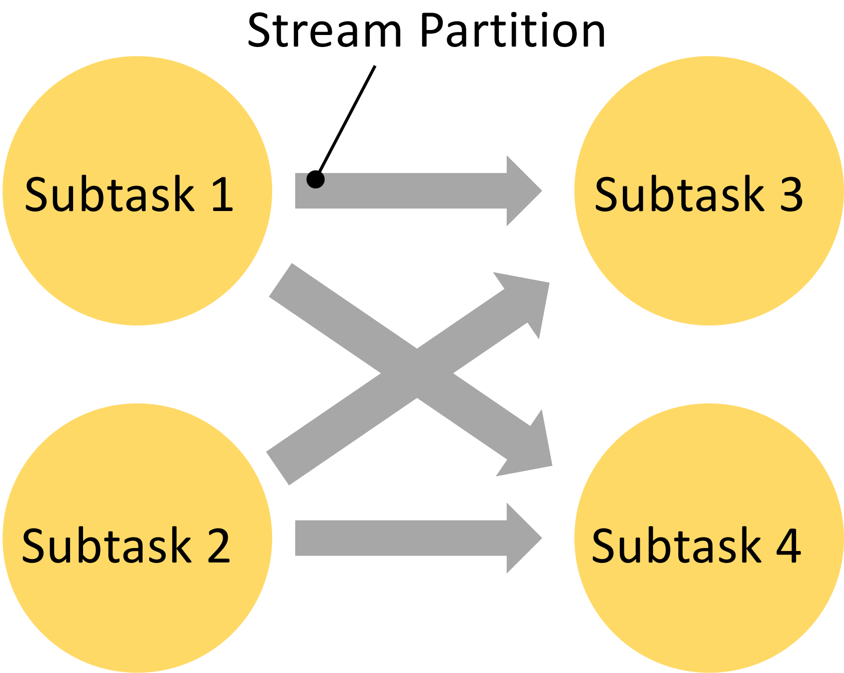
Example of a Stream Processing Application



Anatomy of Stream Processing Application



Stream Partitioning across Subtasks



Abstraction over:

- Subtask output
  - pipelined-bounded
  - pipelined-unbounded
  - Blocking
- Scheduling type
  - all at once
  - next stage on complete output
  - next stage on first output
- Transport
  - high throughput via buffers
  - low latency via buffer timeout

Another Example w/ Flink's DataStream API

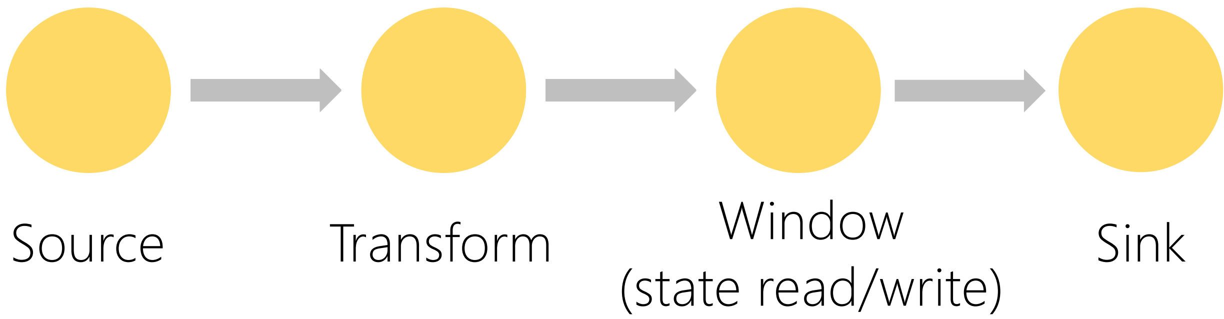
```
val lines: DataStream[String] = env.addSource(new FlinkKafkaConsumer011(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

Source

Transformation

Windowed Transformation

Sink



Streaming Dataflow

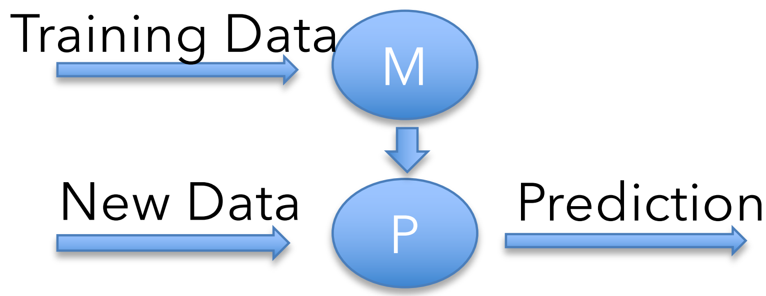
Rich Windowing semantics in Flink

- Trigger policy
  - When to trigger the computation on current window
- Eviction policy
  - When data points should leave the window
  - Defines window width/size
- E.g., count-based policy
  - evict when #elements > n
  - start a new window every n-th element
- Built-in: Count, Time, Delta policies

Flink was the very 1st Open-source framework which supported the Generalized Streaming Model proposed by Google Dataflow/ Apache Beam

Windowing example

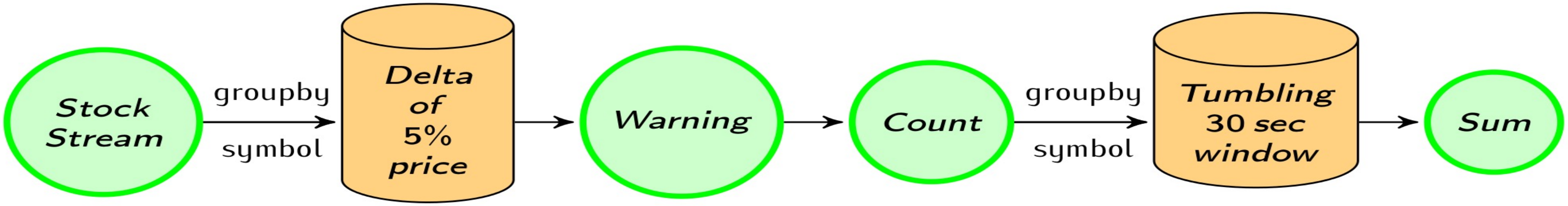
```
//Build new model every minute on the last 5 minutes
//worth of data
val model = trainingData
       .window(Time.of(5,TimeUnit.MINUTES))
       .every(Time.of(1,TimeUnit.MINUTES))
       .reduceGroup(buildModel)

//Predict new data using the most up-to-date model
val prediction = newData
       .connect(model)
       .map(predict);
```



Window Join example

```
case class Name(id: Long, name: String)
case class Age(id: Long, age: Int)
case class Person(name: String, age: Int)

val names = ...
val ages = ...

names.join(ages)
     .onWindow(5, TimeUnit.SECONDS)
     .where("id")
     .equalTo("id") {(n, a) => Person(n.name, a.age)}
```

Yet another example of Stream Processing/ Analysis with Flink




```
case class Count(symbol: String, count: Int)
val defaultPrice = StockPrice("", 1000)

//Use delta policy to create price change warnings
val priceWarnings = stockStream.groupBy("symbol")
  .window(Delta.of(0.05, priceChange, defaultPrice))
  .mapWindow(sendWarning _)

//Count the number of warnings every half a minute
val warningsPerStock = priceWarnings.map(Count(_, 1))
  .groupBy("symbol")
  .window(Time.of(30, SECONDS))
  .sum("count")
```

More at: http://flink.apache.org/news/2015/02/09/streaming-example.html

On Batched vs. Streaming
(The world according to Flink)

A.k.a.: If everything is peachy streams, why is there a DataSet API and where will this end?

A.k.a.: I have heard that "batch is a special case of streaming", so does *<stream processor x>* now own the world?

What changes faster? Data or Query?

Data changes slowly compared to fast changing queries

ad-hoc queries, data exploration, ML training and (hyper) parameter tuning

Batch Processing Use Case

Data changes fast application logic is long-lived

continuous applications, data pipelines, standing queries, anomaly detection, ML evaluation, …

Stream Processing Use Case

Summary on Another View of Batched vs. Streaming

What Changes Fasters ? Your Code or Your Data ?

- dData/dt >> dCode/dt => a Data Streaming problem
- dCode/dt >> dData/dt => a Data Exploration problem

(and likely to become a Data Streaming problem later)

Src: Prof. Joe Hellerstein of UCBerkeley

What changes faster? Data or Query?

Data changes slowly compared to fast changing queries

ad-hoc queries, data exploration, ML training and (hyper) parameter tuning

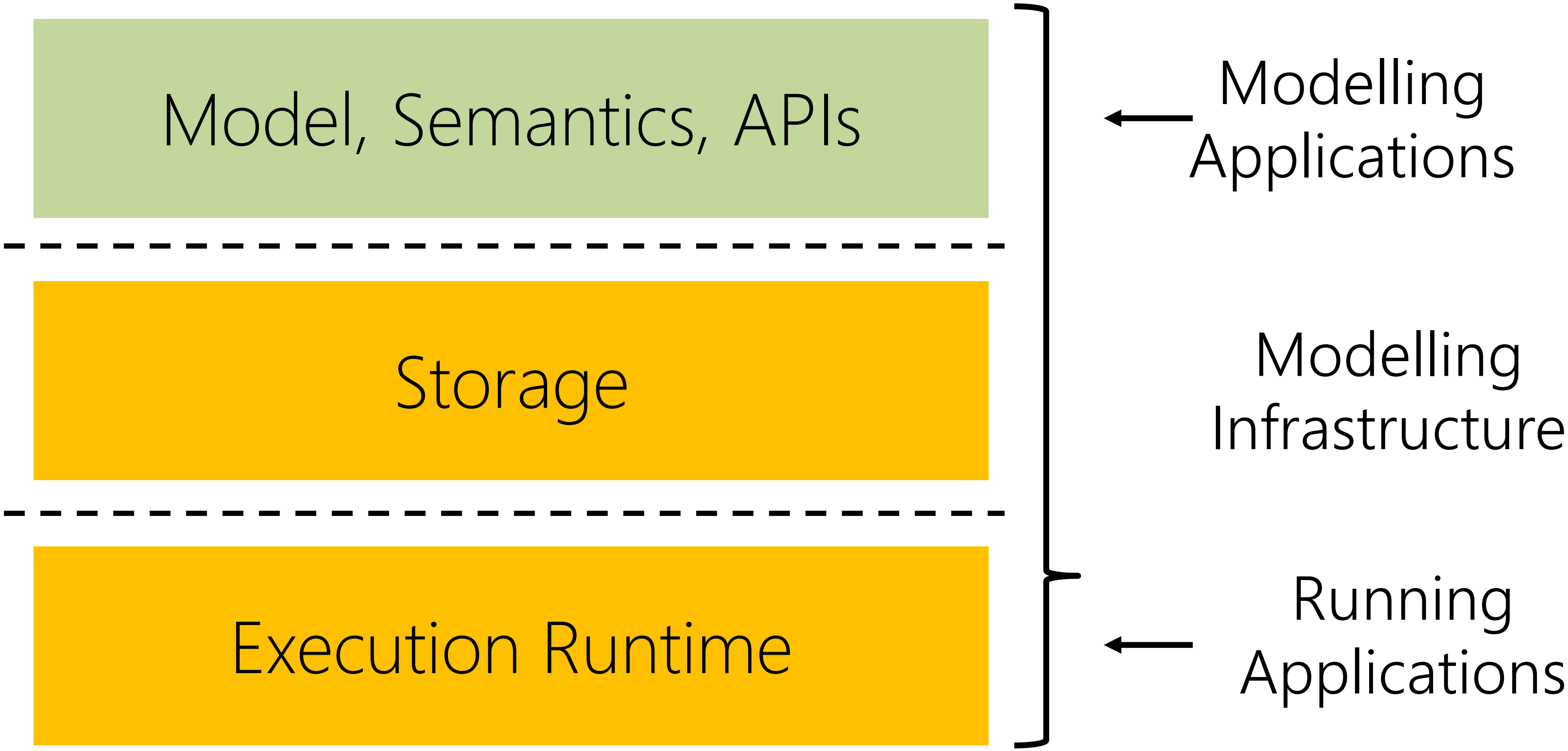
DataSet API

Data changes fast application logic is long-lived

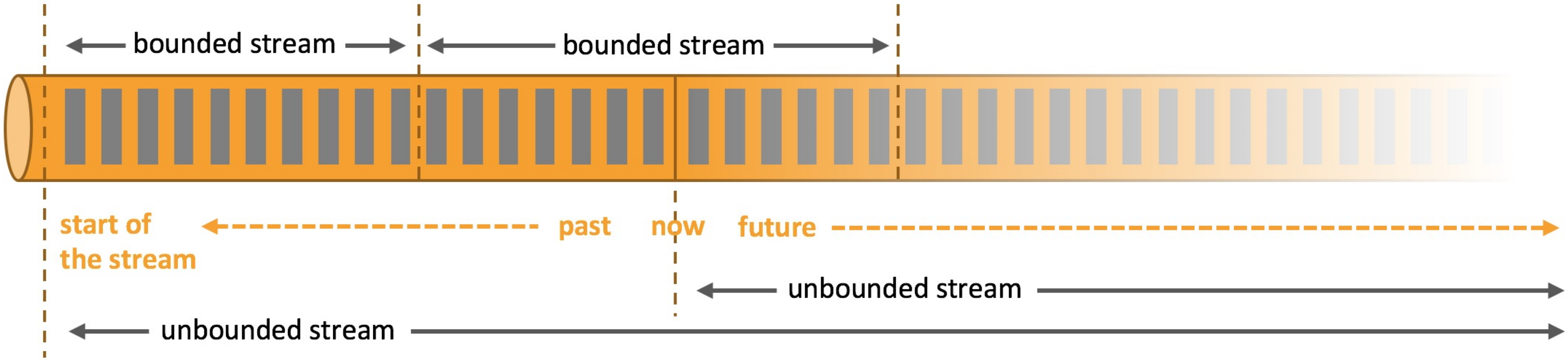
continuous applications, data pipelines, standing queries, anomaly detection, ML evaluation, …

DataStream API

Abstraction/APIs and Runtime



Samentics/APIs: Everything Streams



Flink is good here…

Eventual goal of Flink, Not yet achieved as of Feb 2019

Data changes slowly compared to fast changing queries

Data changes fast application logic is long-lived

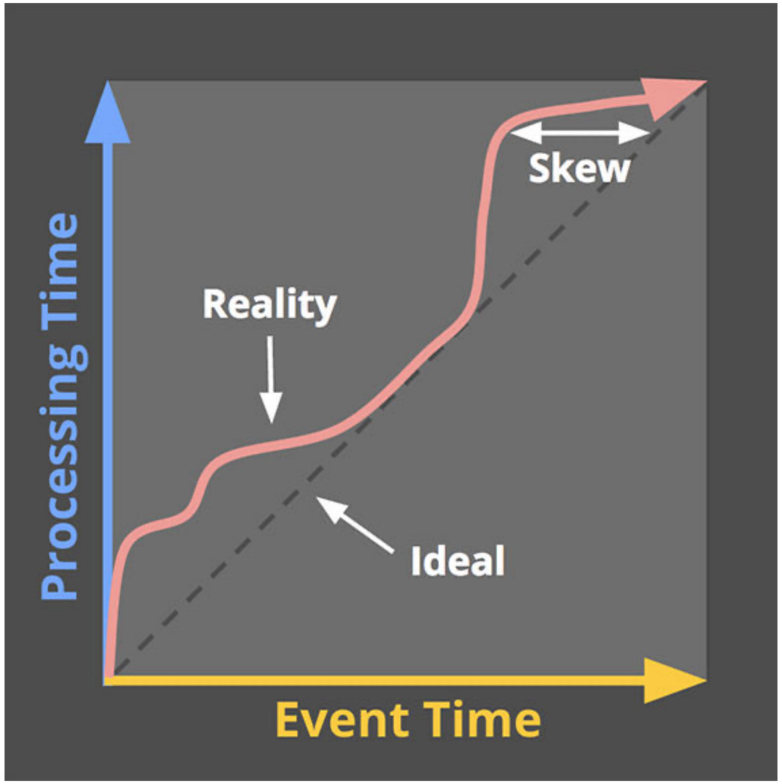
DataStream API

BoundedStream

DataStream API

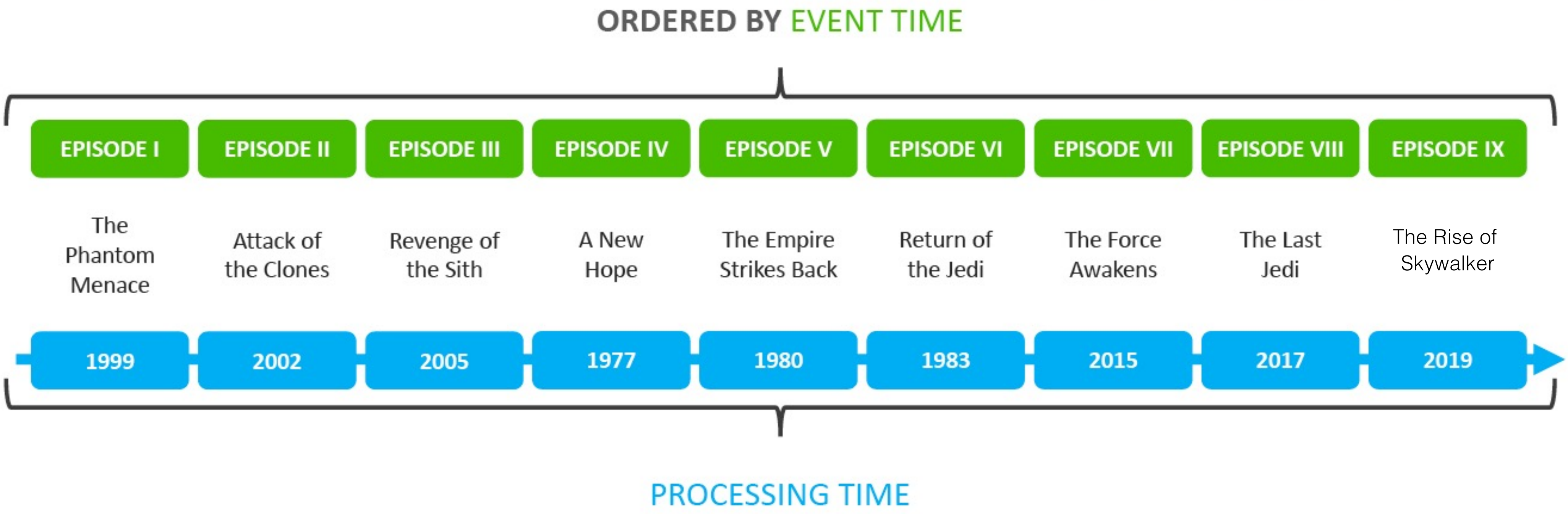
UnboundedStream

Latency vs. Completeness *(in Tyler's words)*



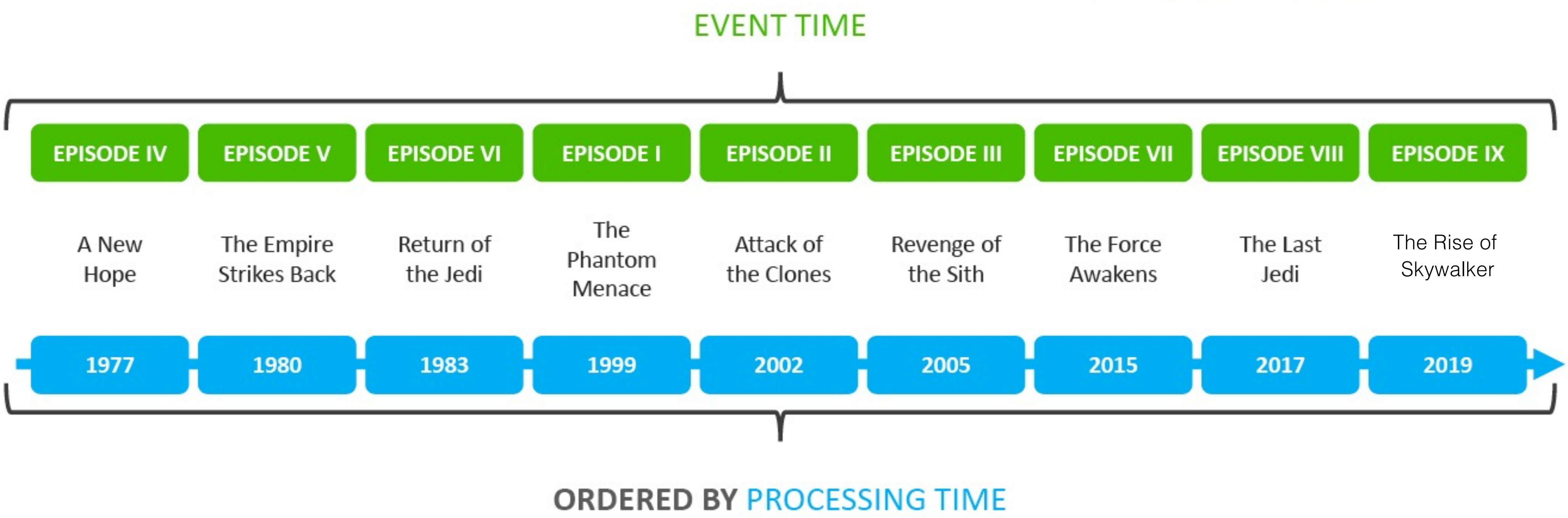
Latency vs. Completeness

TIME IN STREAMING



Latency vs. Completeness

TIME IN STREAMING



Latency versus Completeness

Bounded/ Batch	Unbounded/ Streaming
Data is as complete as it gets within that Batch Job No fine latency control	Trade of latency versus completeness

The Eventual Goal of Flink (WIP as of Apr 2021)

Data changes slowly compared to fast changing queries

ad-hoc queries, data exploration, ML training and (hyper) parameter tuning

DataSet API

Data changes fast application logic is long-lived

continuous applications, data pipelines, standing queries, anomaly detection, ML evaluation, …

DataStream API

The Eventual Goal of Flink (WIP as of Apr 2021)

Data changes slowly compared to fast changing queries

ad-hoc queries, data exploration, ML training and (hyper) parameter tuning

Dat API

Data changes fast application logic is long-lived

continuous applications, data pipelines, standing queries, anomaly detection, ML evaluation, …

DataStream API

The Eventual Goal of Flink (WIP as of Apr 2021)

Data changes slowly compared to fast changing queries

DataStream API

BoundedStream

Data changes fast application logic is long-lived

DataStream API

UnboundedStream

The Eventual Goal of Flink (WIP as of Apr 2021)

DataStream API

BoundedStream

No latency SLA

Assume Data Completeness

DataStream API

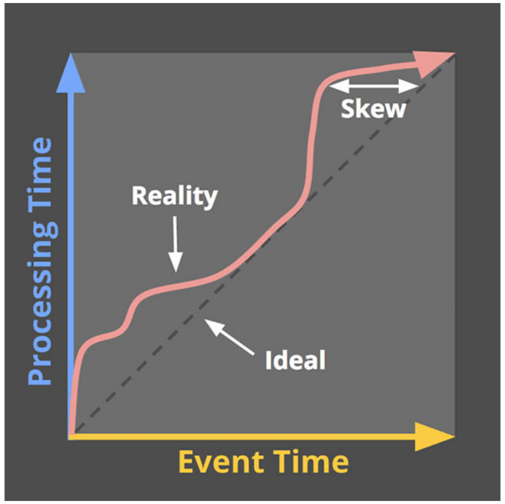
UnboundedStream

Latency / Completeness Tradeoff

On the Runtime Side?

Streaming

- Keep up with real time, some extra capacity for catch-up
- Receive data roughly in order as produced
- Latency is important



Batch

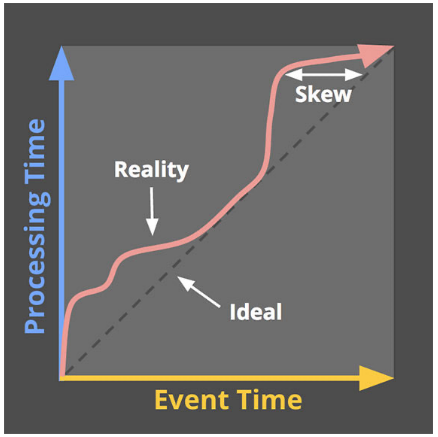
- Fast forward through months/years of history
- Massively parallel unordered reads
- Throughput most important

Streaming Runtime

- Time in data stream must be quasi monotonous, produce time progress (watermarks)

- Always have close-to-latest incremental results

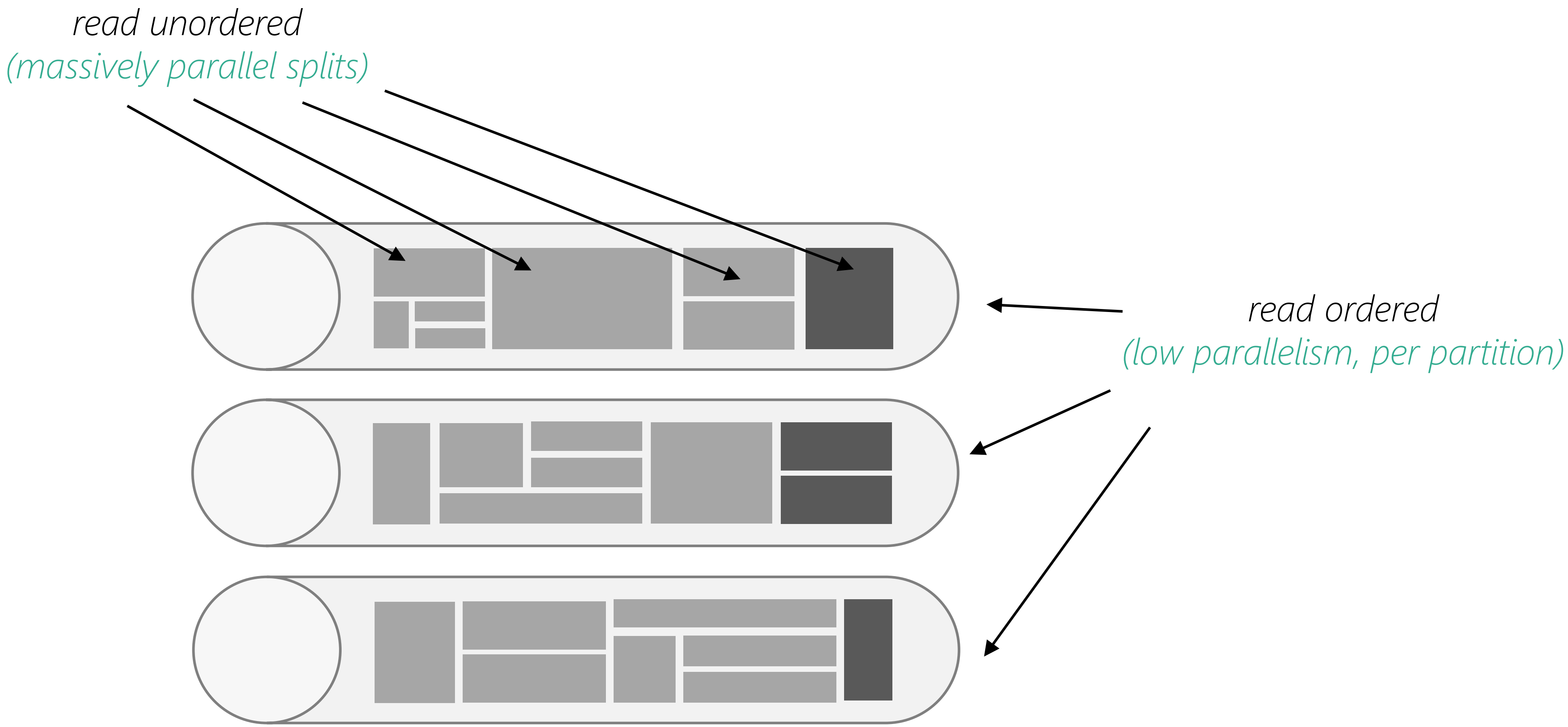
- Resource needs change over time



Batch Runtime

- Order of time in data does not matter (parallel unordered reads)
- Bulk operations (2 phase hash/sort)
- Longer time for recovery (no low latency SLA)
- Resource requirements change fast throughout the execution of a single job

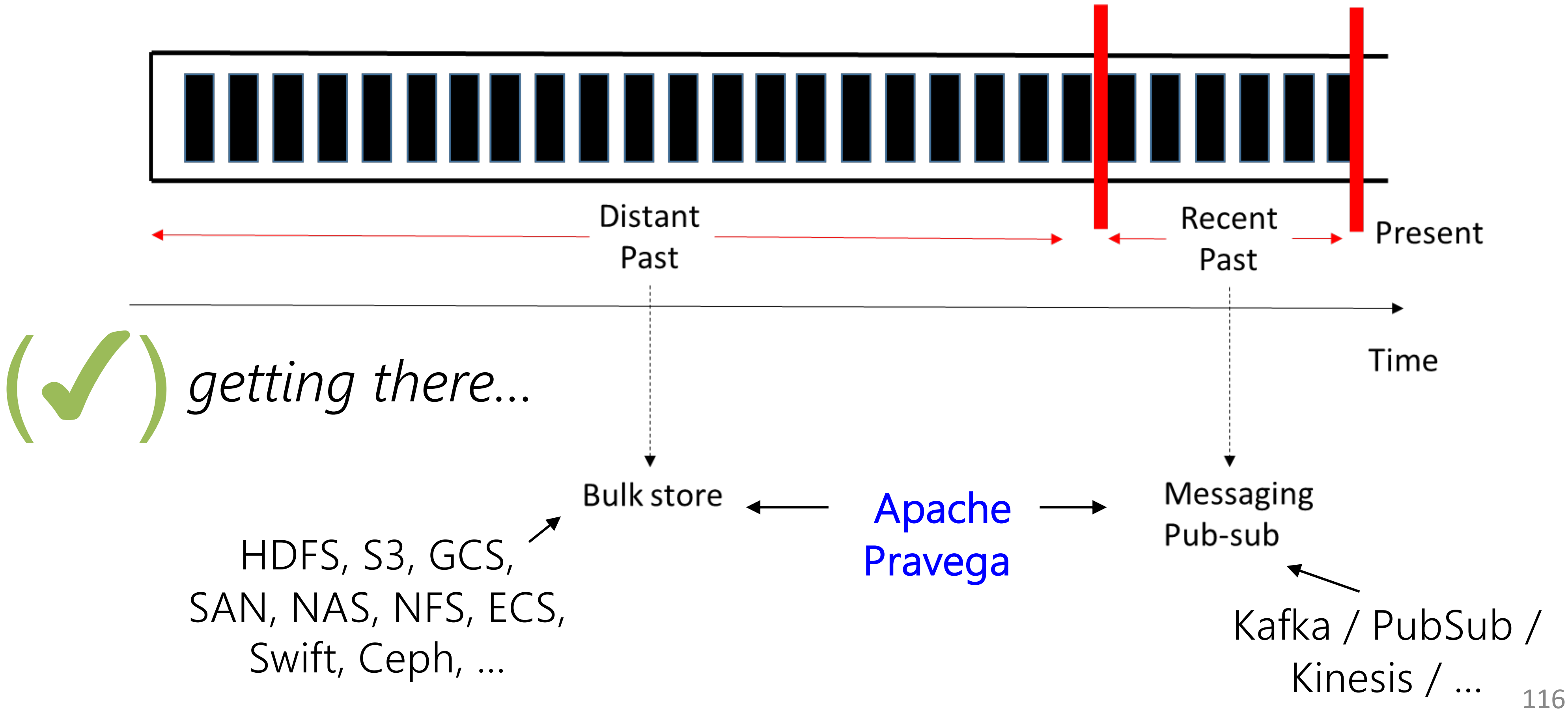
Ordered and unordered reads



What is Flink's take here?

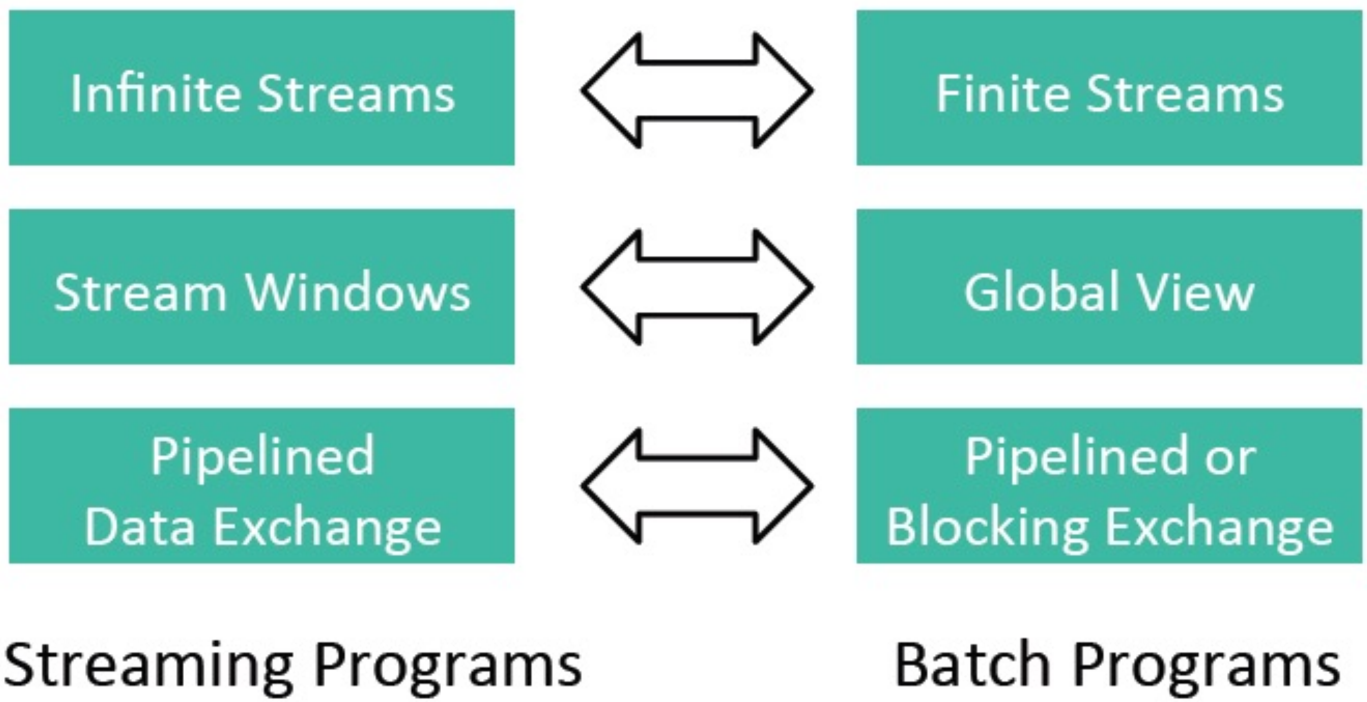
- Unique Network Stack, high throughput, low latency, memory speed
- Unique Fault Tolerance Model that recovers batch and streaming with tunable cost / recovery-lag
- Sources can read streams and parallel input splits
- Different Data Structures optimized for incremental results (DataStream API) and for batch results (DataSet API)
- Most unified runtime, but more unification in Runtime still needed...

Streams and Storage

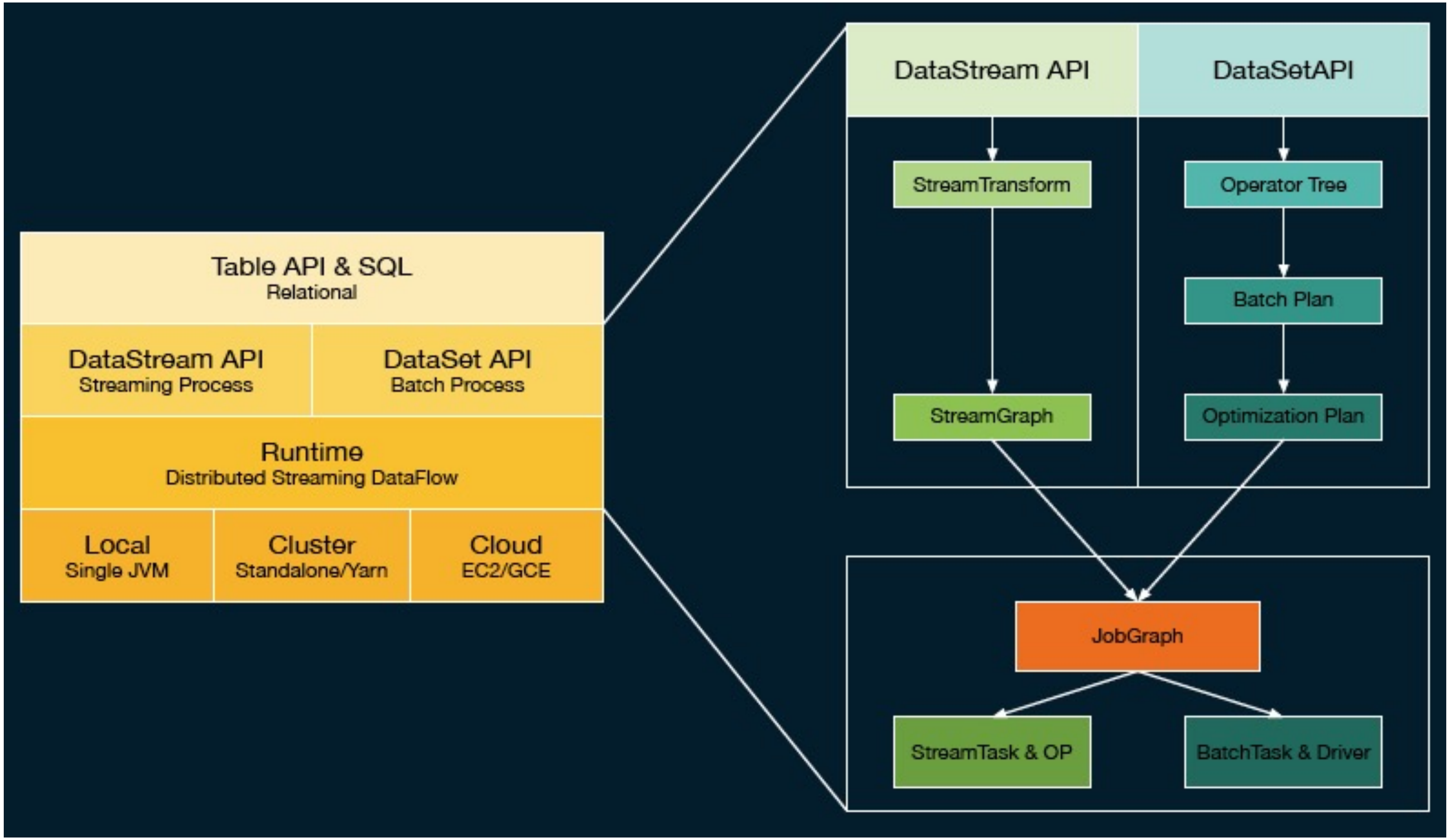


Summary of Batch on Streaming

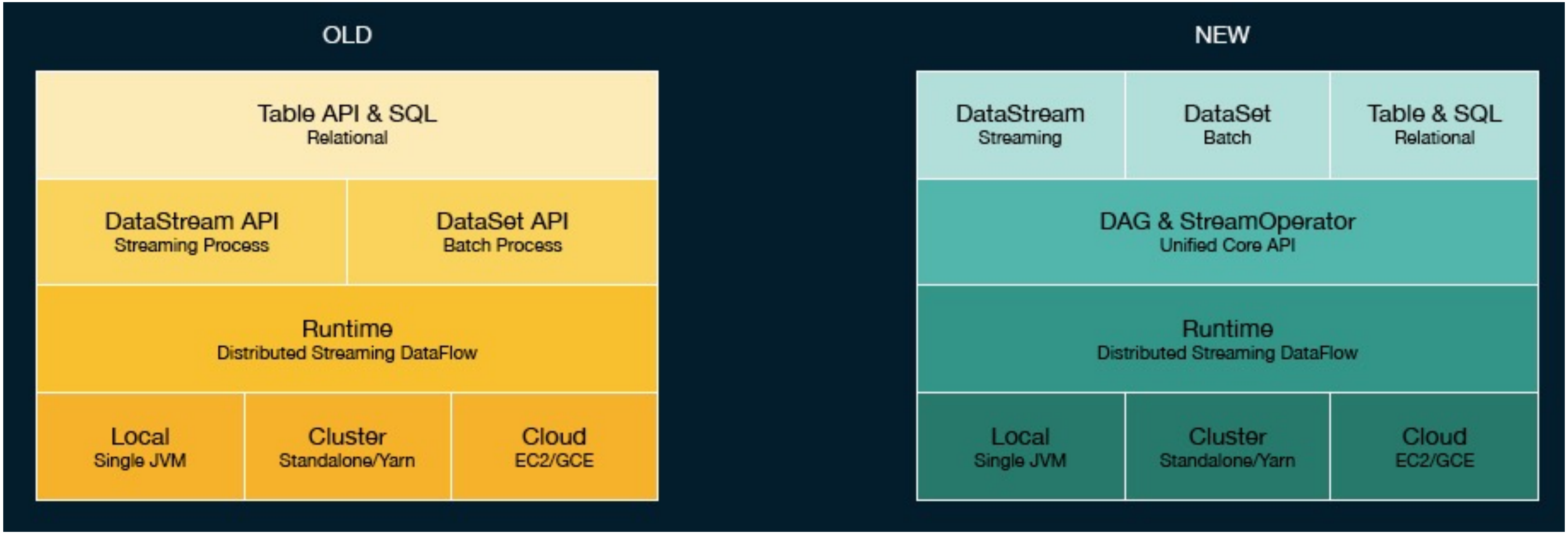
- Batch programs are a special kind of streaming program



Current Flink API Stack (circa 1Q2019)

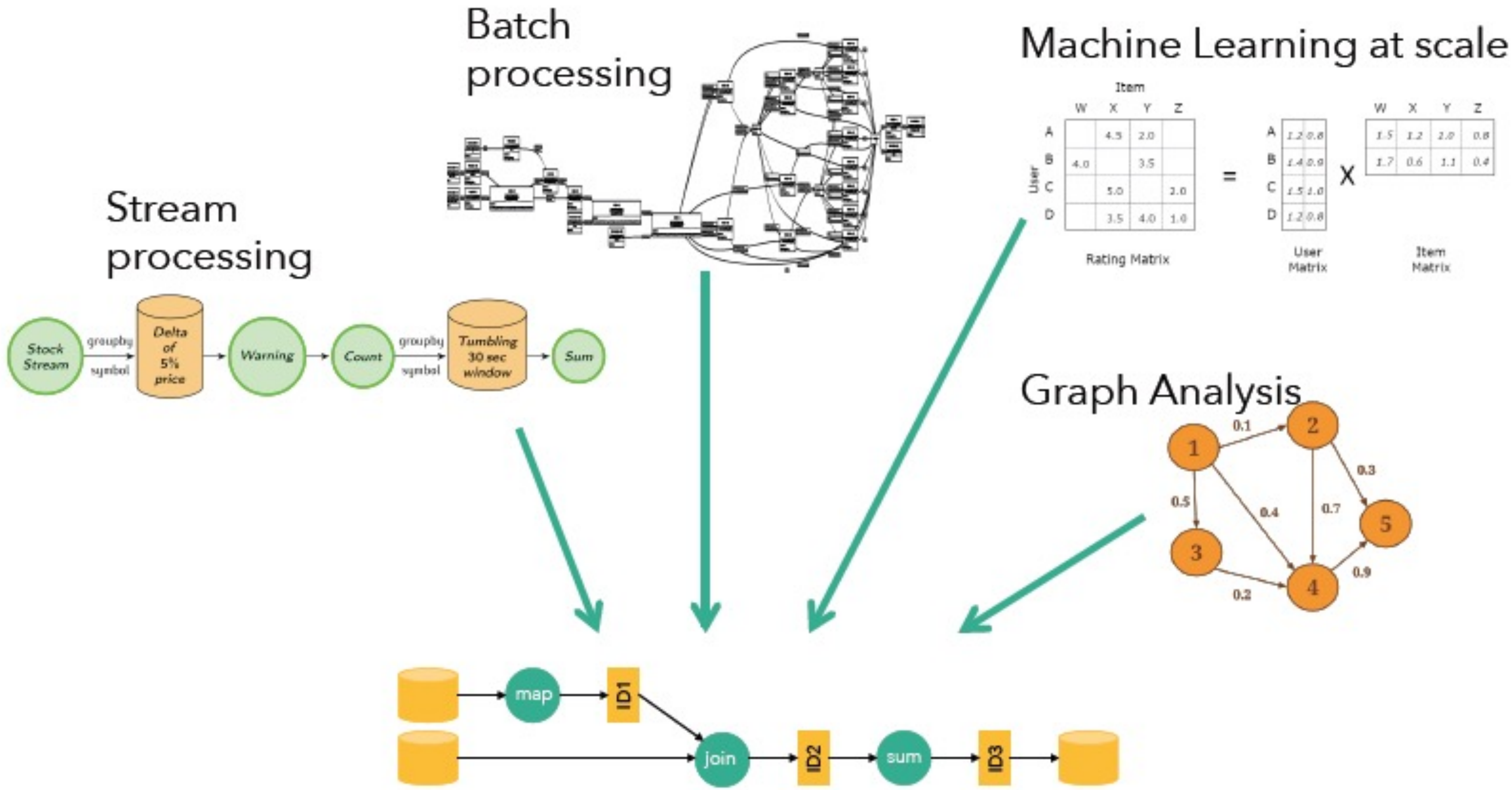


Proposed New Flink API Stack (WIP)



- https://flink.apache.org/news/2019/02/13/unified-batch-streaming-blink.html

Goal: Stream Processor for all Applications

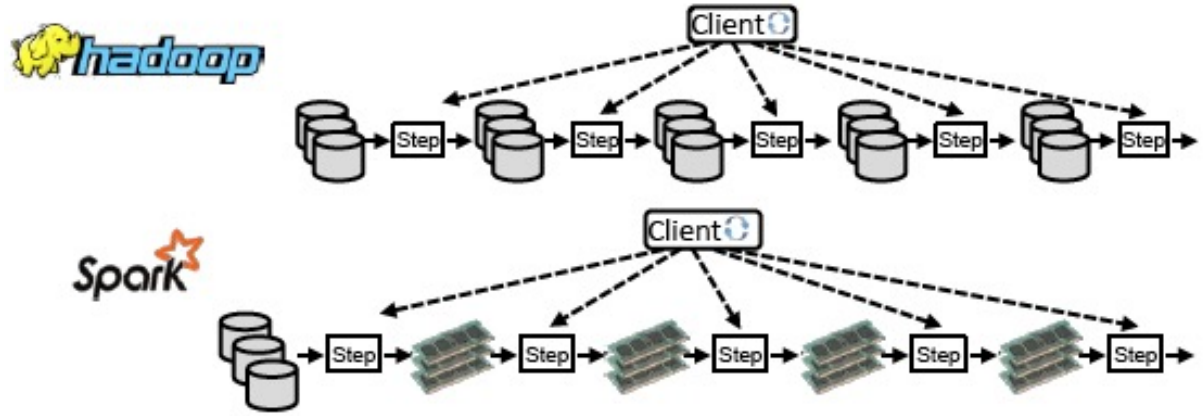


Gelly – Flink's Graph Library (on its way out)

- Library with graph operations
  - Common graph stats, PageRank, SSSP, Connected Components, label propagation
  - Vertex-centric API
  - Gather-apply-scatter API

Flink's Native Support for Iteration

Built-in vs. Driver-based Looping (Iteration) Support

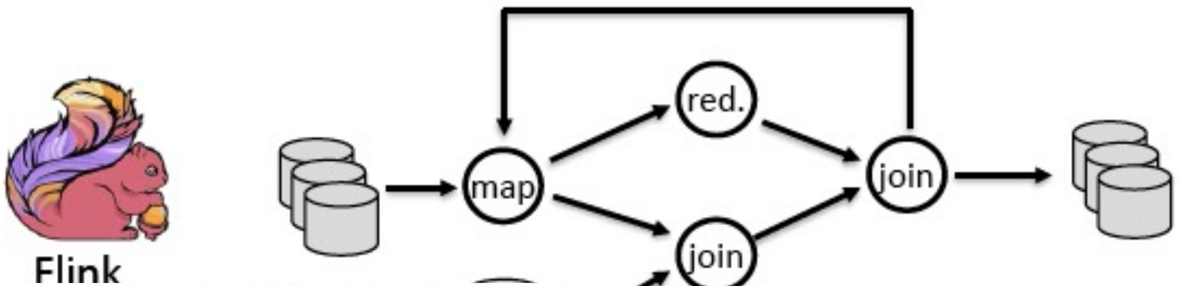


Loop outside the system, in driver program

For/While loop in client submits one job per iteration step

=> iterative program looks like many independent jobs

Data reuse by caching in memory and/or disk



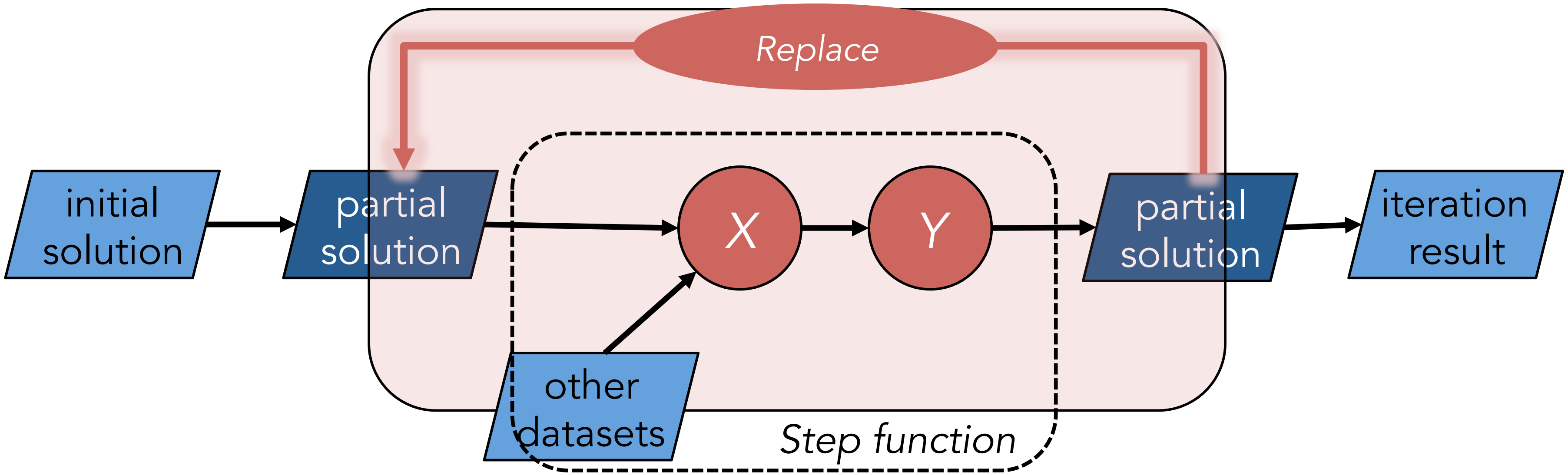
Dataflows with feedback edges

System is iteration-aware, can optimize the job

S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, V. Markl: Iterative Parallel Data Processing with Stratosphere: an Inside Look. SIGMOD 2013

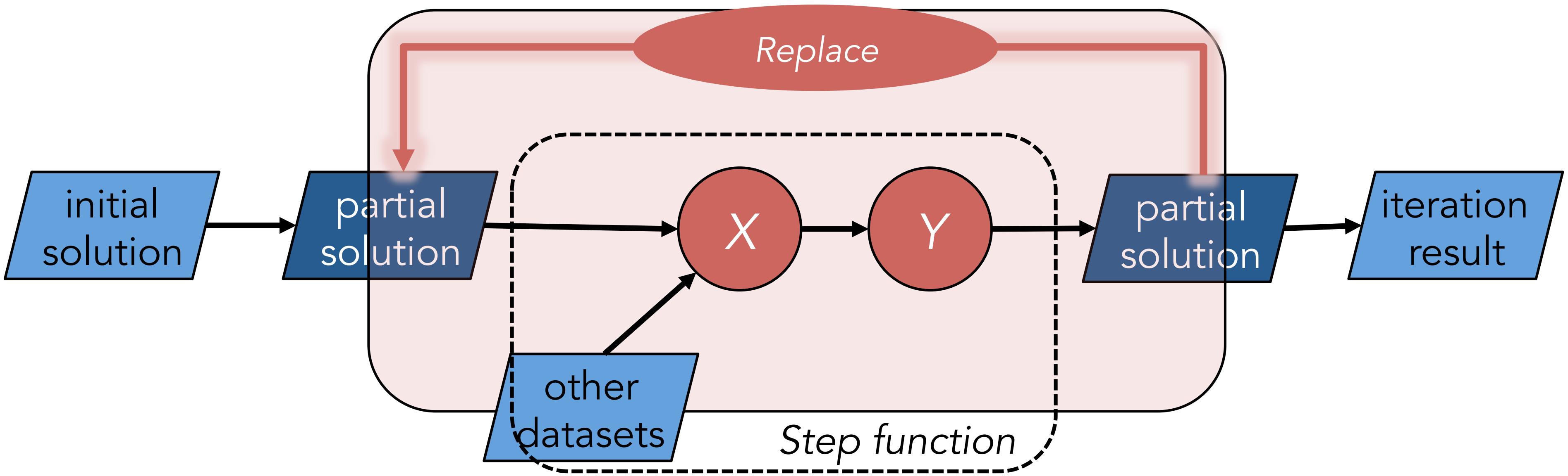
S. Ewen, K. Tzoumas, M. Kaufmann, V. Markl: Spinning Fast Iterative Data Flows. PVLDB 5(11): 1268-1279 (2012)

Flink supports iterations in the Dataflow



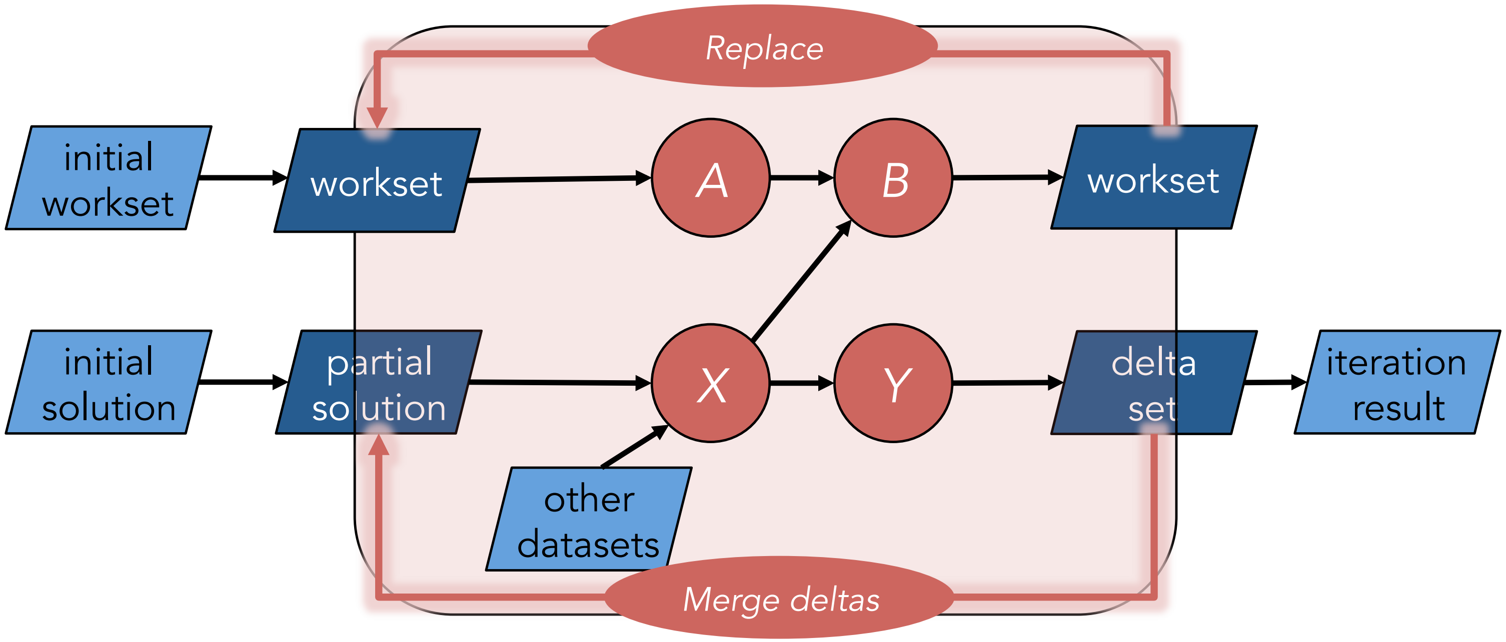
- Built-in operator to support looping over data
- Apply Step-function to partial solution until convergence
- Step-function can be arbitrary Flink program
- Convergence via fixed number of iterations or custom convergence criteria.
- Operator state is preserved across different iterations
- Loop-invariant data is cached

Flink supports Iterations in the Dataflow



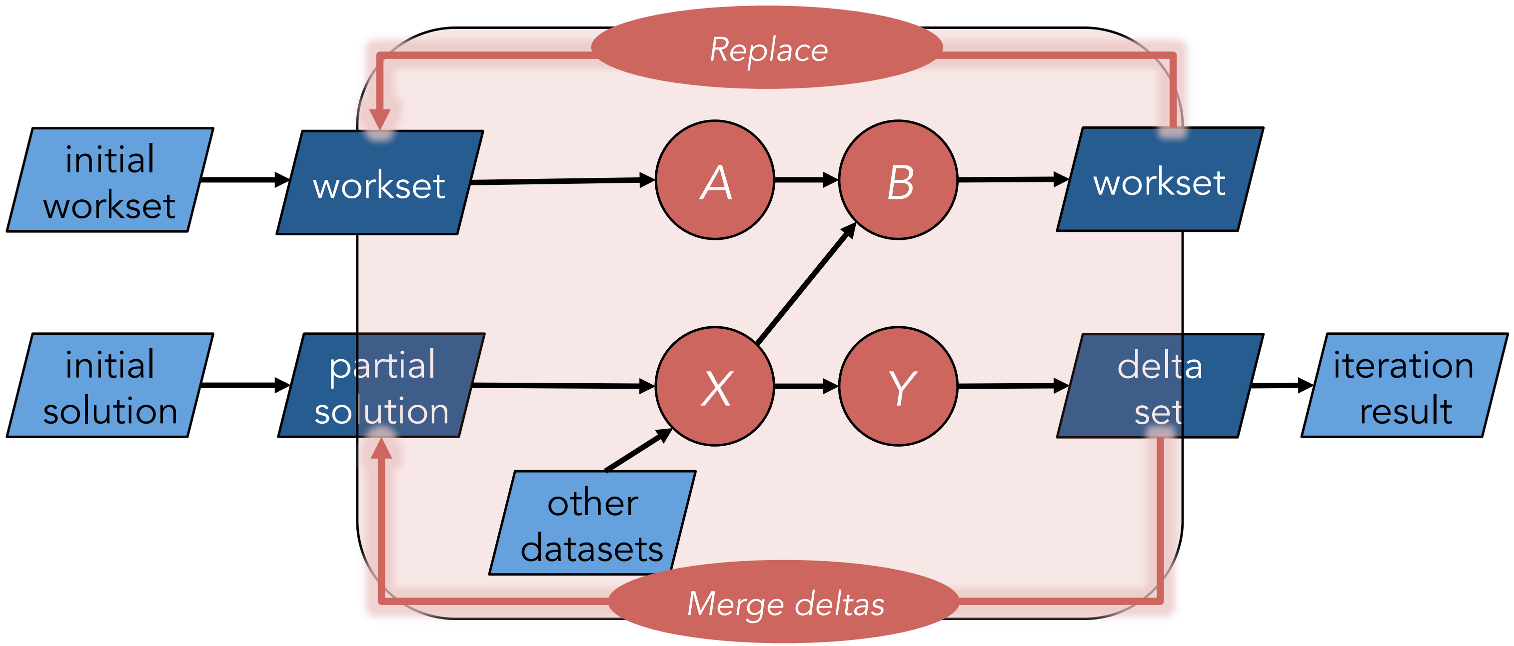

```
DataSet<Page> pages = ...
DataSet<Neighborhood> edges = ...

IterativeDataSet<Page> pagesIter = pages.iterate(maxIterations);
DataSet<Page> newRanks = update (pagesIter, edges);
DataSet<Page> result = pagesIter.closeWith(newRanks)
```

Iterate natively with deltas (i.e. Stateful Iterations)



- Compute next workset and changes to partial solution until workset is empty.
- Generalize vertex-centric computing model of Pregel and Graphlab
- Efficient and fits well with Graph-based algorithms and ML applications

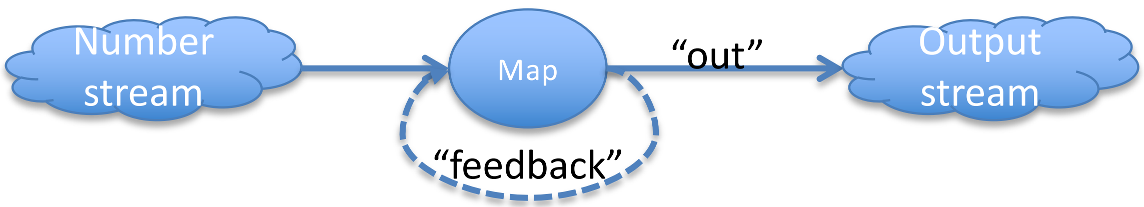
Iterate natively with deltas (i.e. Stateful Iterations)



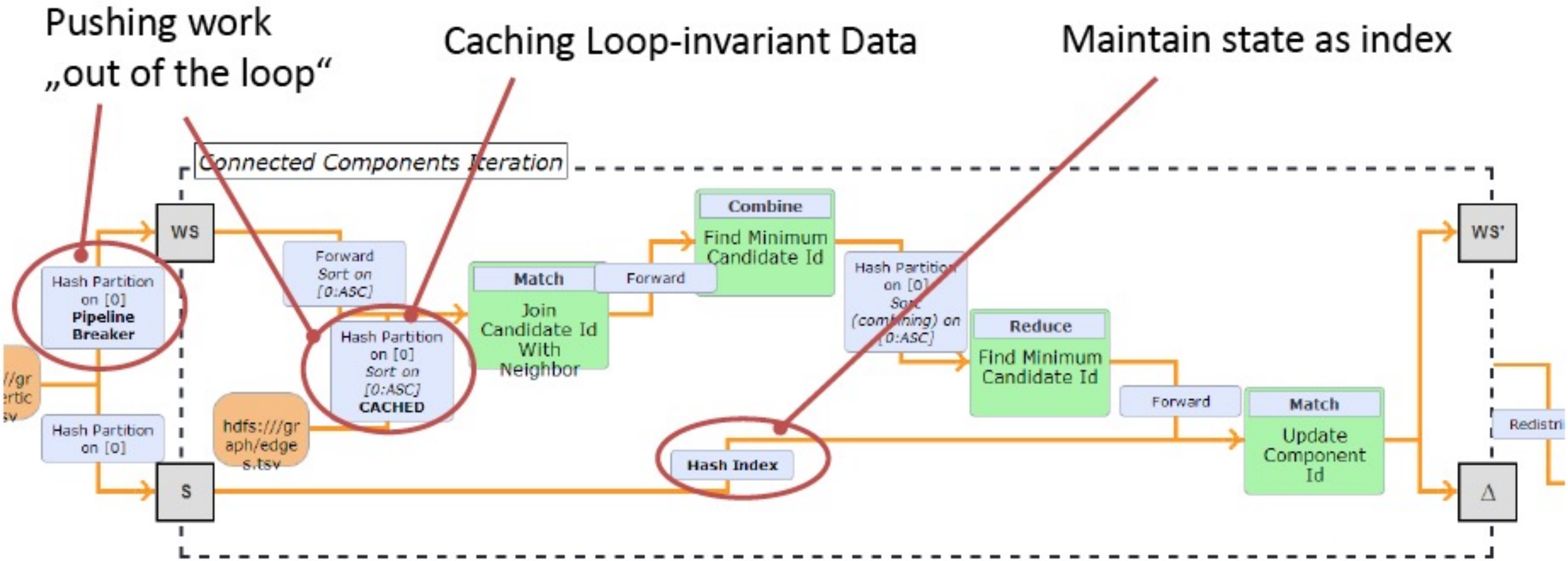

```
DeltaIteration<...> pagesIter = pages.iterateDelta(initialDeltas, maxIterations, 0);
DataSet<...> newRanks = update (pagesIter, edges);
DataSet<...> newRanks = ...
DataSet<...> result = pagesIter.closeWith(newRanks, deltas)
```

Iterative processing example

```scala
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.generateSequence(1, 10).iterate(incrementToTen, 1000).print

env.execute("Iterative example")

def incrementToTen(input: DataStream[Long]) = {
    val incremented = input.map {_ + 1}
    val split = incremented.split
          {x => if (x >= 10) "out" else "feedback"}
    (split.select("feedback"), split.select("out"))
  }
```



Optimizing Iterative Programs



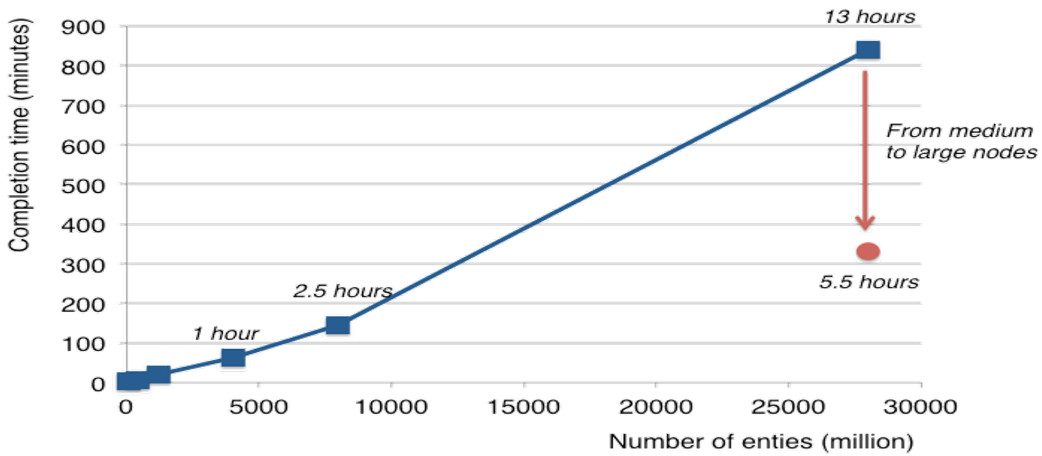
Yet another Example: Iterative processing

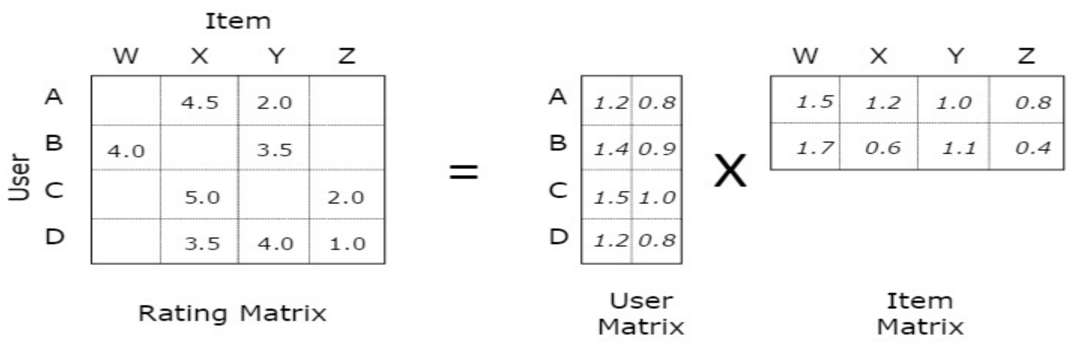
```
DataSet<Page> pages = ...
DataSet<Neighborhood> edges = ...
DataSet<Page> oldRanks = pages; DataSet<Page> newRanks;

for (i = 0; i < maxIterations; i++) {
  newRanks = update(oldRanks, edges)
  oldRanks = newRanks
}
DataSet<Page> result = newRanks;
```

```
DataSet<Page> update (DataSet<Page> ranks, DataSet<Neighborhood> adjacency) {
  return oldRanks
    .join(adjacency)
    .where("id").equalTo("id")
    .with ( (page, adj, out) -> {
      for (long n : adj.neighbors)
        out.collect(new Page(n, df * page.rank / adj.neighbors.length))
    })
    .groupBy("id")
    .reduce ( (a, b) -> new Page(a.id, a.rank + b.rank) );
```

An Example (ML application) which needs Iterations in the Dataflow

Factorizing a matrix with 28 billion ratings for recommendations

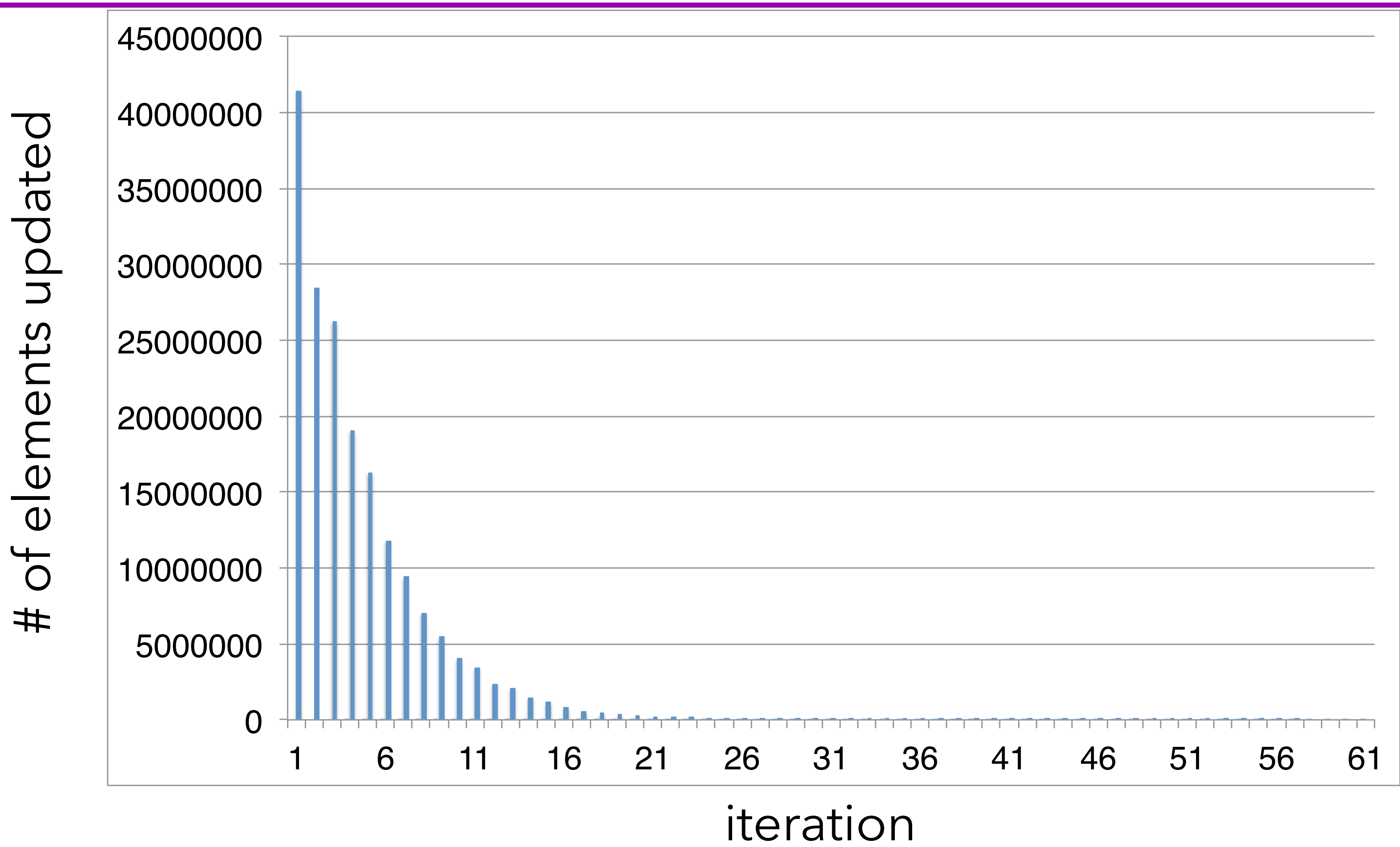




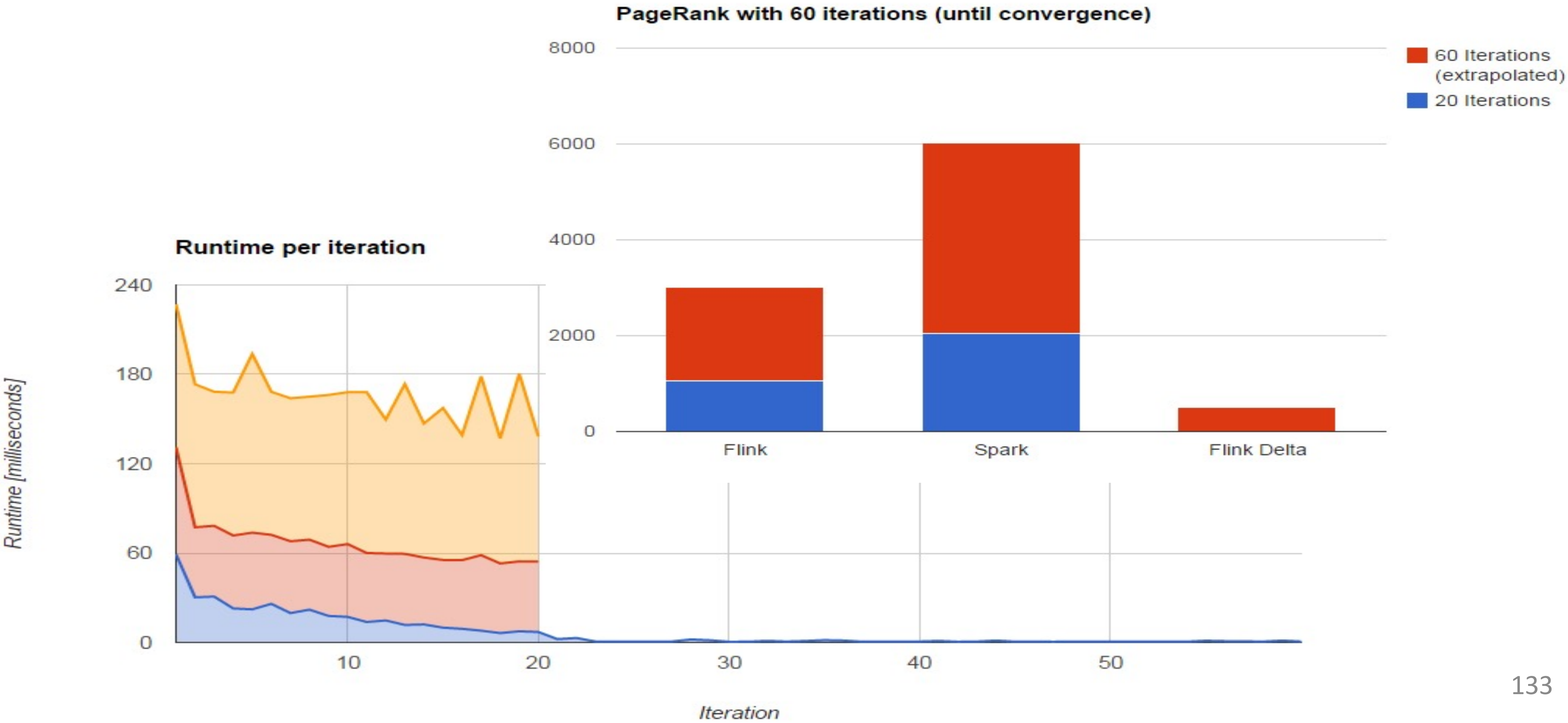
(Scale of Netflix or Spotify)

More at: http://data-artisans.com/computing-recommendations-with-flink.html

Benefits with Delta Iterations

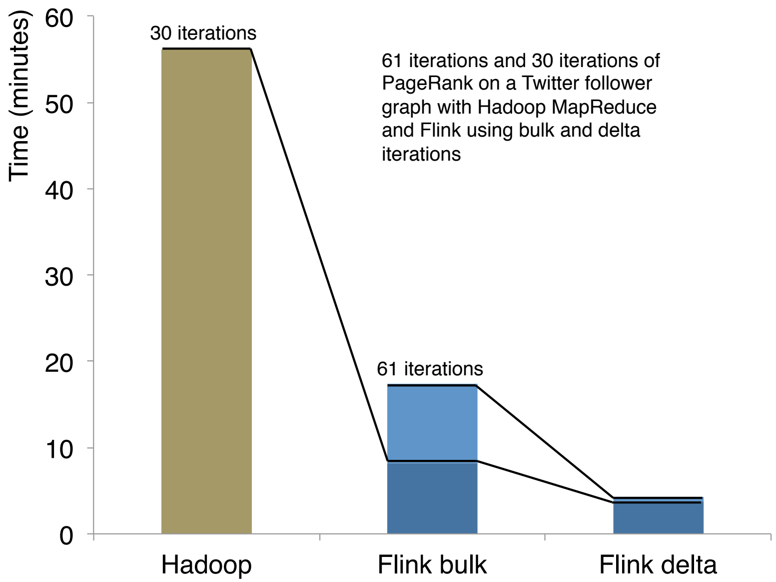


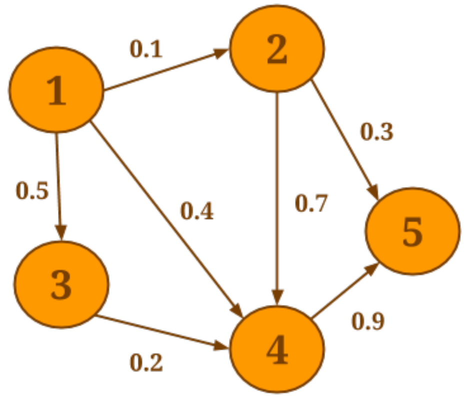
Performance Comparison b/w Native, Unrolling, and Delta



Delta Iterations => Fast Graph analysis etc

Performance competitive with dedicated graph analysis systems





… and mix and match ETL-style and graph analysis in one program

More at: http://data-artisans.com/data-analysis-with-flink.html

Other API elements & tools

- Accumulators and counters
  - Int, Long, Double counters
  - Histogram accumulator
  - Define your own
- Broadcast variables
- Visualization
- Local debugging/testing mode

Recall: Layered Abstractions of Flink

Layered abstractions to
navigate simple to complex use cases

```sql
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp)
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

High-level Analytics API — **Stream SQL / Tables** *(dynamic tables)*

Stream- & Batch Data Processing — **DataStream API** *(streams, windows)*

```scala
val stats = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum((a, b) -> a.add(b))
```

Stateful Event-Driven Applications — **Process Function** *(events, state, time)*

```scala
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // work with event and state
  (event, state.value) match { … }

  out.collect(…) // emit events
  state.update(…) // modify state

  // schedule a timer callback
  ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
}
```

Low Level: Process Function

```java
public void processElement1(Transaction txn, Context ctx, Collector<Transaction> out) {
    // keep the transaction in the internal state until the approval comes
    pendingTransaction.update(txn);
    // schedule a timer to trigger the timeout
    ctx.timerService().registerProcessingTimeTimer(txn.getTimestamp() + TIMEOUT_MILLIS);
}

public void processElement2(ApproveOrReject approval, Context ctx, Collector<Transaction> out) {
    // get and remove the transaction from the state
    Transaction txn = pendingTransaction.value();
    pendingTransaction.clear();
    // forward the transaction to the main stream
    out.collect(txn);
}

public void onTimer(long timestamp, OnTimerContext ctx, Collector<Transaction> out) {
    // check if the transaction is still there, in which case it would be timed out
    Transaction txn = pendingTransaction.value();
    if (txn != null) {
        // write to the timeout stream
        ctx.output(TIMEOUT_STREAM, txn);
        pendingTransaction.clear();
    }
}
```

Strength of DataStream API

- Very expressive stream processing
  - Transform data, update state, define windows, aggregate, etc
- Highly customizable windowing logic
  - Assigners, Triggers, Evictors, Lateness
- Asynchronous I/O
  - Improve communication to external systems
- Low-level operations

Limitations of DataStream API

- Writing Distributed programs is not easy
  - Stream processing technology spreads/changes rapidly
  - New Streaming concepts (time, state, …)

- Require knowledge & skill
  - Continuous applications have special requirements
  - Programming experience (Java/ Scala)

=> Learning curve can be steep

- Most users want to focus on their business logic

Design Goals for Flink Table & SQL API

- Easy, Declarative and concise Relational API
- Tool for a wide range of use cases
- Unification of Batch & Streaming with SAME semantics
- Queries efficiently executed
  - Let Flink handle state, time, and common mistakes

Apache Flink’s Relational API

ANSI SQL

```
SELECT user, COUNT(url) AS cnt
FROM clicks
GROUP BY user
```

LINQ-style Table API

```
tableEnvironment
  .scan("clicks")
  .groupBy('user)
  .select('user, 'url.count as 'cnt)
```

Unified APIs for batch & streaming data

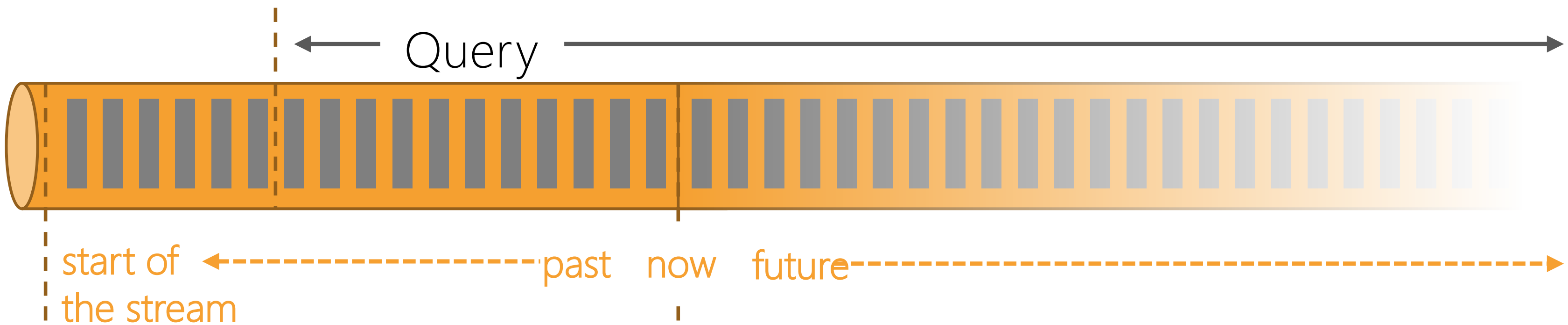
A query specifies exactly the same result regardless whether its input is static batch data or streaming data.

Another Example of Table API

```
val customers = envreadCsvFile(…).as('id, 'mktSegment)
      .filter( 'mktSegment === "AUTOMOBILE" )

val orders = env.readCsvFile(…)
      .filter( o => dateFormat.parse(o.orderDate).before(date) )
      .as('orderId, 'custId, 'orderDate, 'shipPrio)

val items = orders
      .join(customers).where('custId === 'id)
      .join(lineitems).where('orderId === 'id)
      .select('orderId,'orderDate,'shipPrio,
          'extdPrice * (Literal(1.0f) - 'discount) as 'revenue)

val result = items
      .groupBy('orderId, 'orderDate, 'shipPrio)
      .select('orderId, 'revenue.sum, 'orderDate, 'shipPrio)
```

High Level: SQL *(ANSI)*

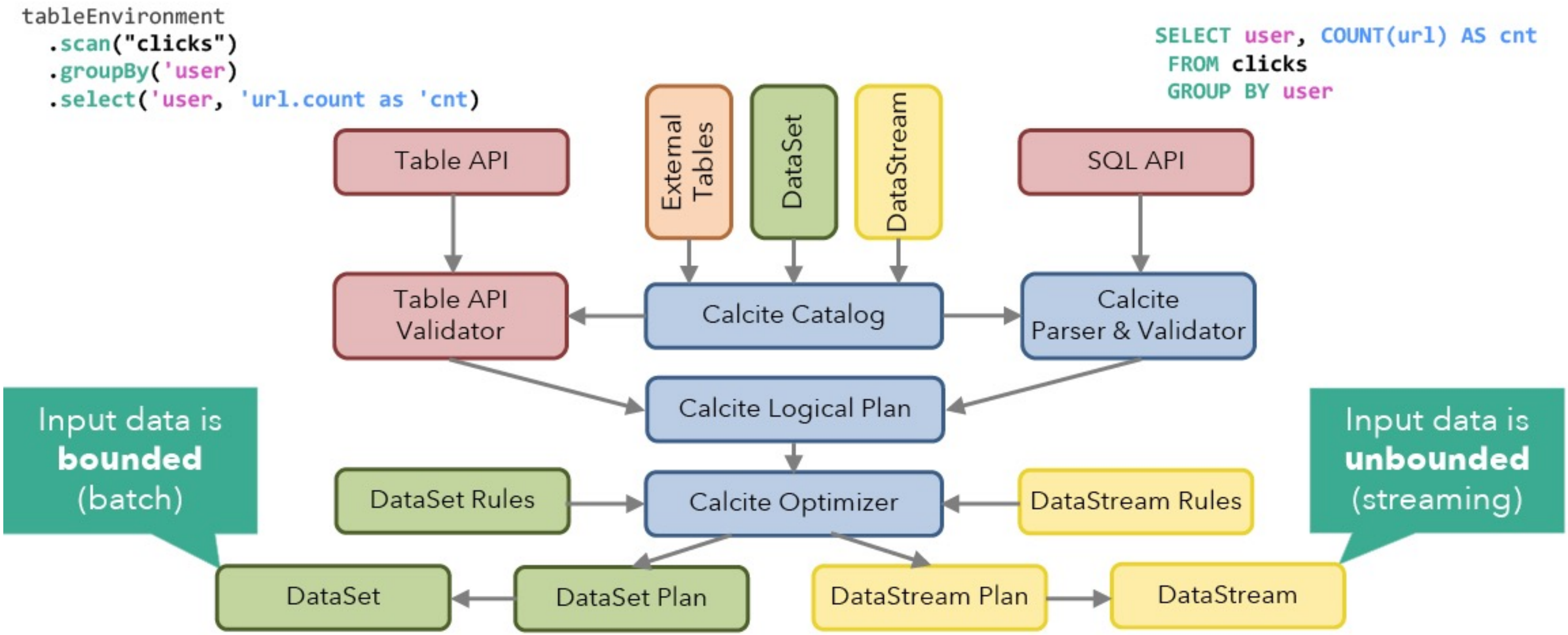
```
SELECT
  campaign,
  TUMBLE_START(clickTime, INTERVAL ’1’ HOUR),
  COUNT(ip) AS clickCnt
FROM adClicks
WHERE clickTime > ‘2017-01-01’
GROUP BY campaign, TUMBLE(clickTime, INTERVAL ‘1’ HOUR)
```



Features supporting Data Pipelines

- Support for POJOs, maps, arrays, and other nested types
- Large set of built-in functions (150+)
  - LIKE, EXTRACT, TIMESTAMPADD, FROM_BASE64, MD5, STDDEV_POP, AVG, …
- Support for custom UDFs (scalar, table, aggregate)

Query Translation



What if "Clicks" is a File ?

Clicks

Input data is read at once

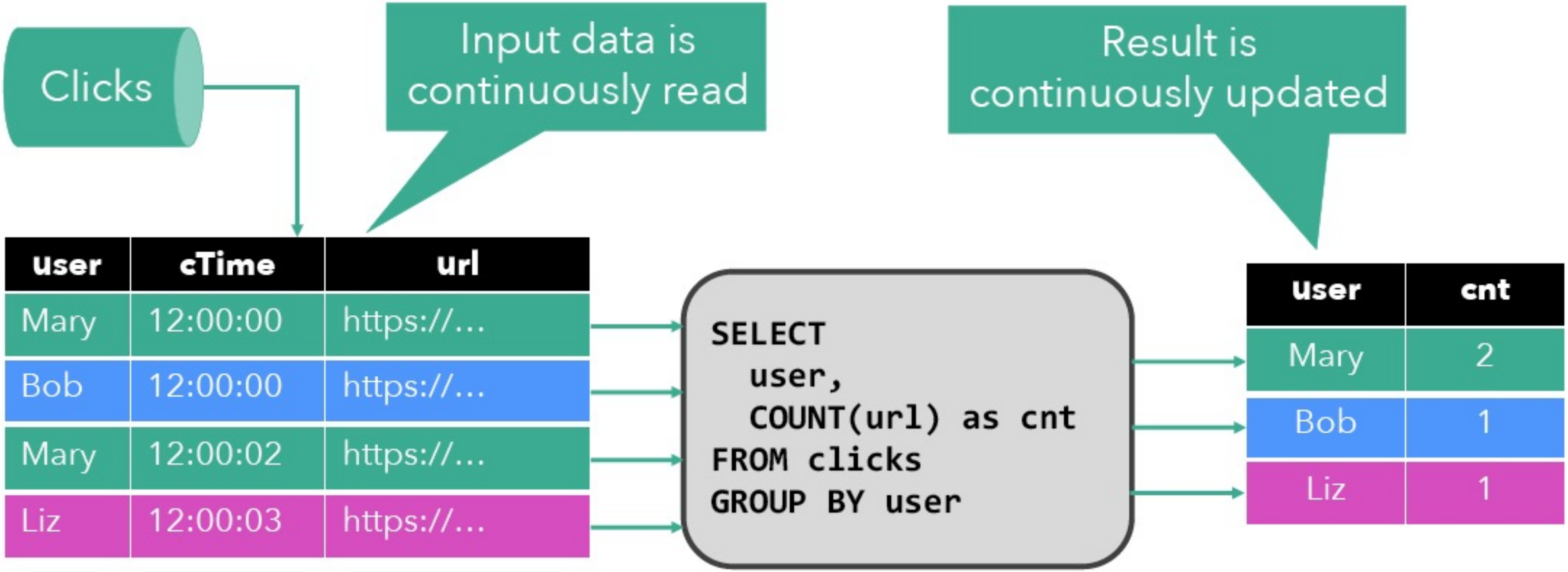
user	cTime	url
Mary	12:00:00	https://...
Bob	12:00:00	https://...
Mary	12:00:02	https://...
Liz	12:00:03	https://...

```
SELECT
  user,
  COUNT(url) as cnt
FROM clicks
GROUP BY user
```

Result is produced at once

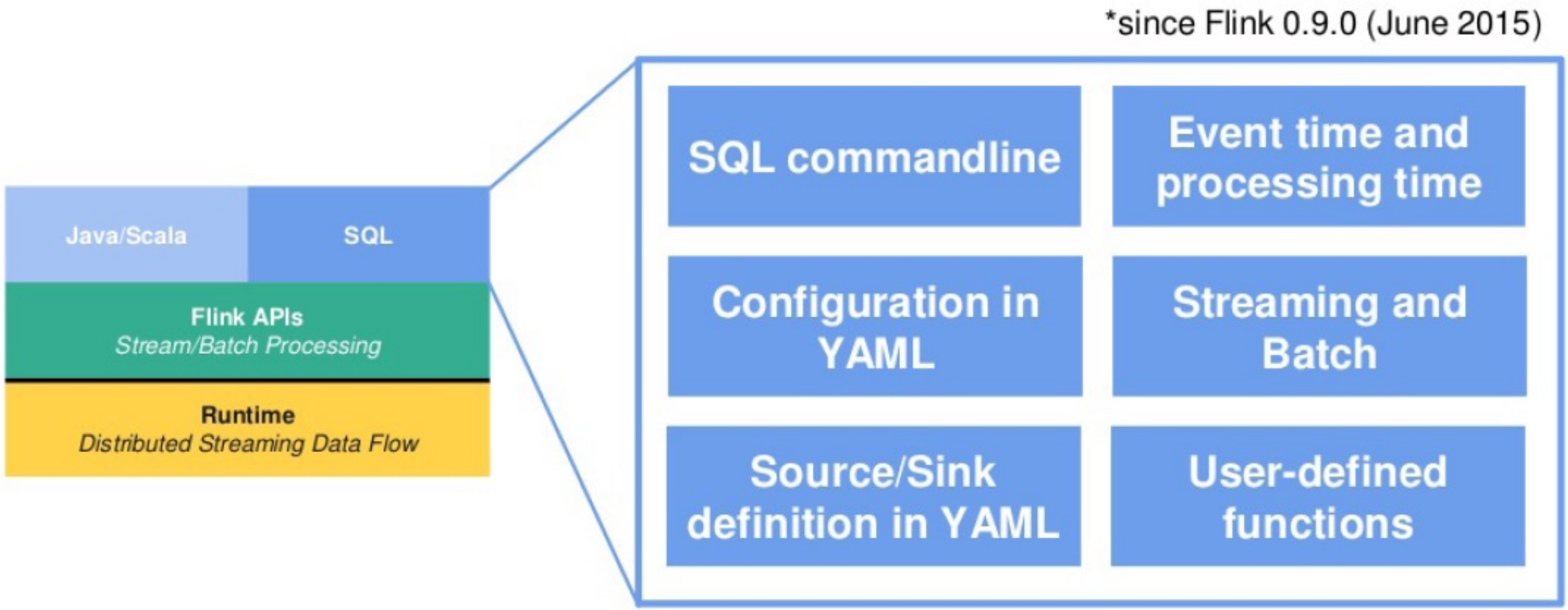
user	cnt
Mary	2
Bob	1
Liz	1

What if "Clicks" is a Stream ?



The result is the same!

Flink SQL

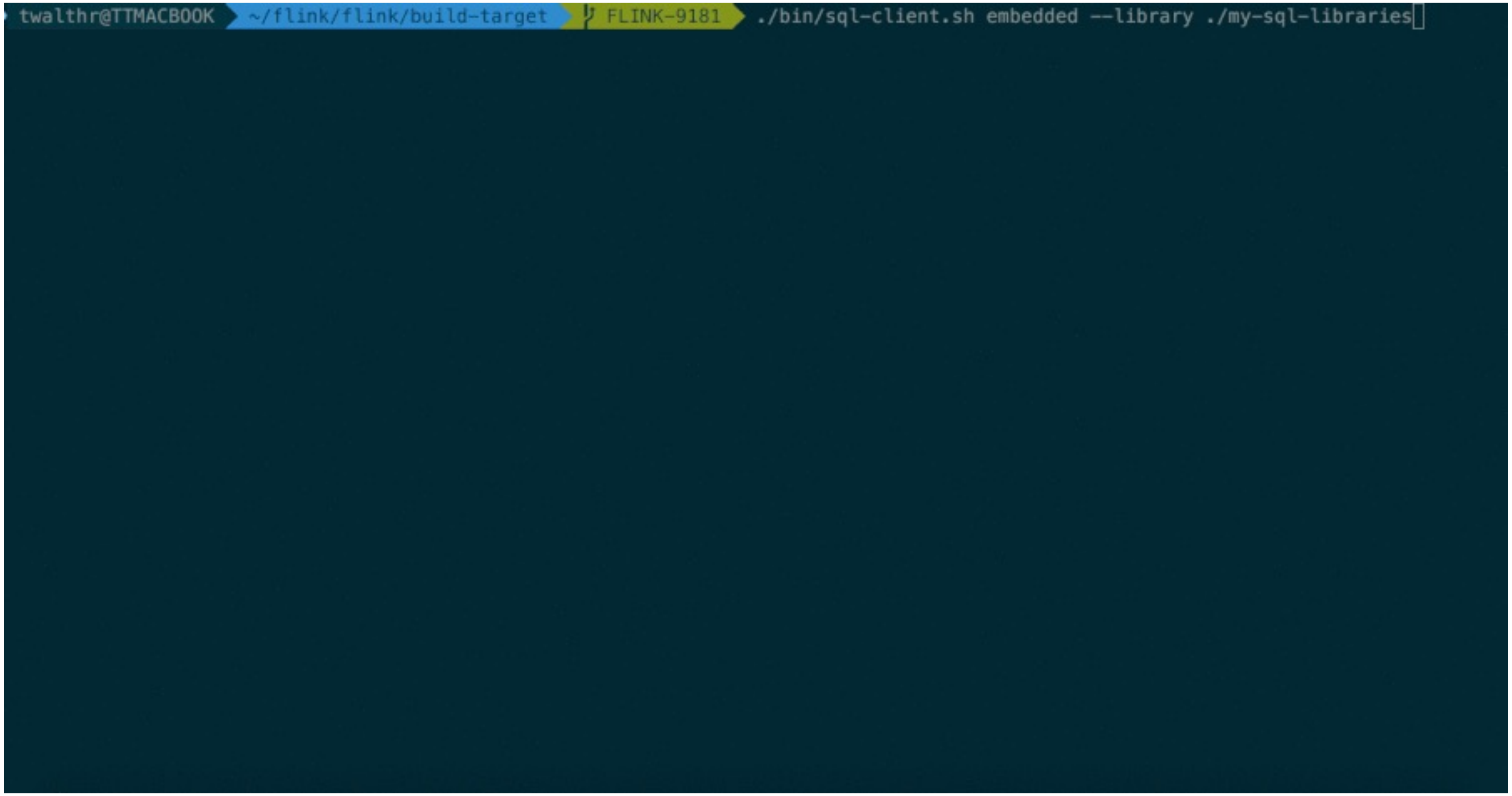


“NO CODING REQUIRED”

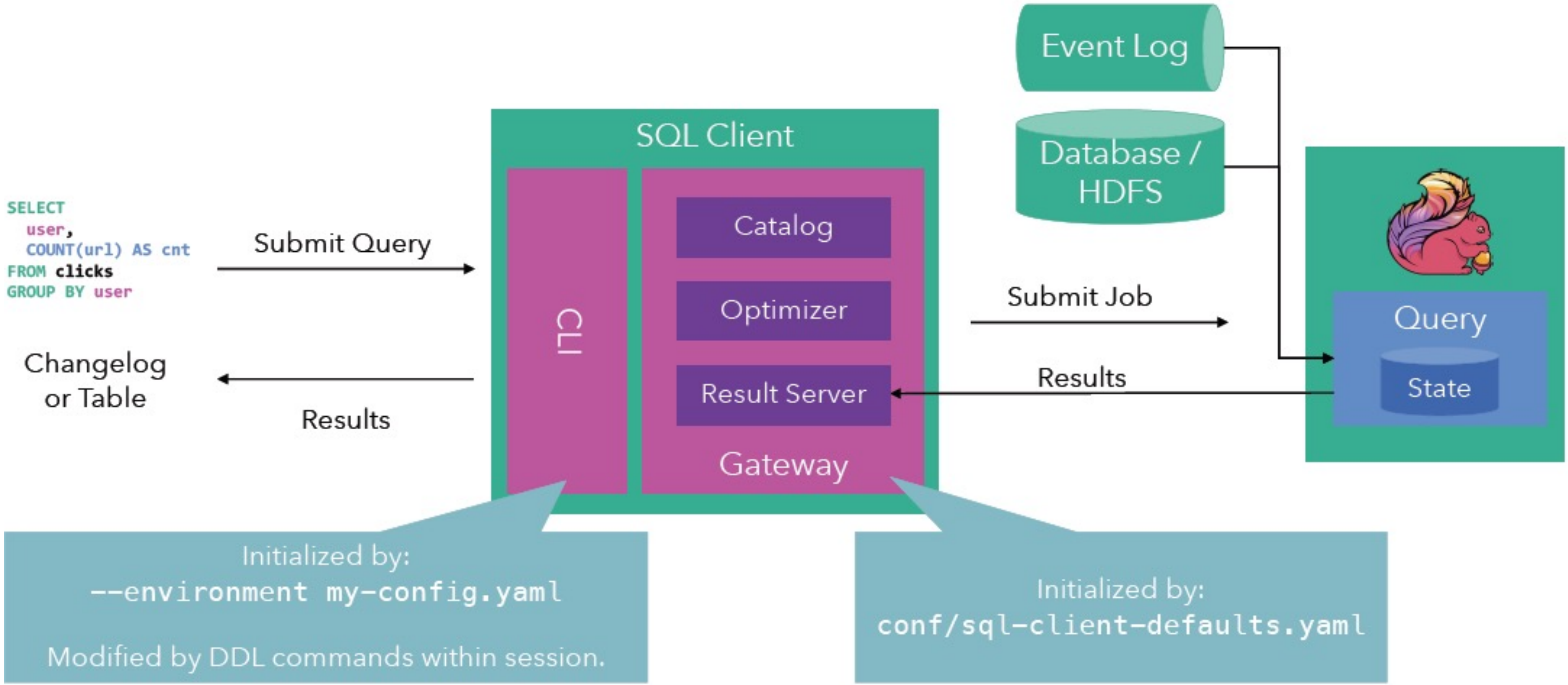
SQL Feature set in Flink 1.6.0

- SELECT FROM WHERE
- GROUP BY / HAVING
  - Non-windowed, TUMBLE, HOP, SESSION windows
- JOIN / IN
  - Windowed INNER, LEFT / RIGHT / FULL OUTER JOIN
  - Non-windowed INNER, LEFT / RIGHT / FULL OUTER JOIN
- [streaming only] OVER / WINDOW
  - UNBOUNDED / BOUNDED PRECEDING
- [batch only] UNION / INTERSECT / EXCEPT / ORDER BY

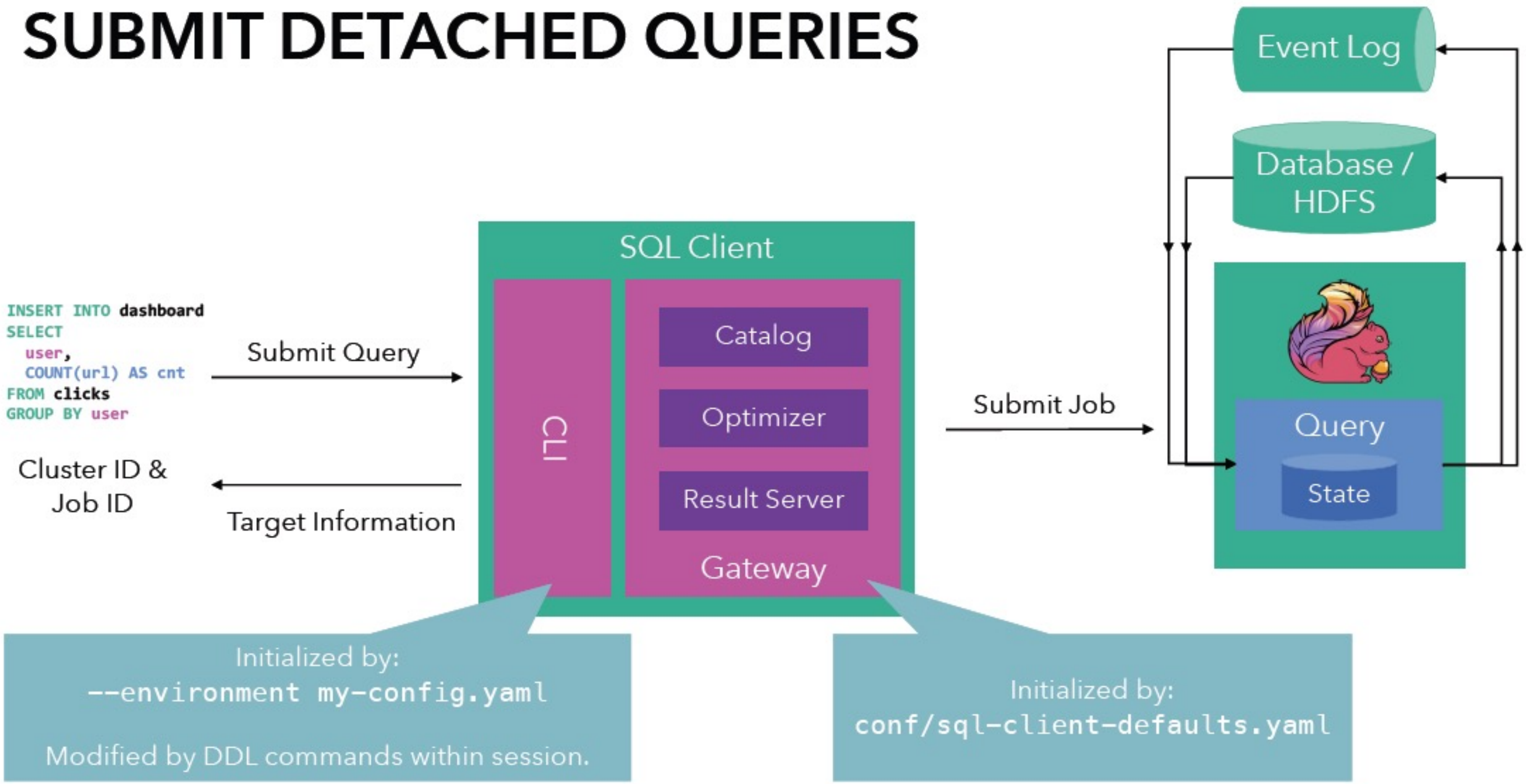
SQL Client


twalthr@TTMACBOOK ~/flink/flink/build-target FLINK-9181 ./bin/sql-client.sh embedded --library ./my-sql-libraries

How to use Flink SQL



SUBMIT DETACHED QUERIES



Extended JOIN support

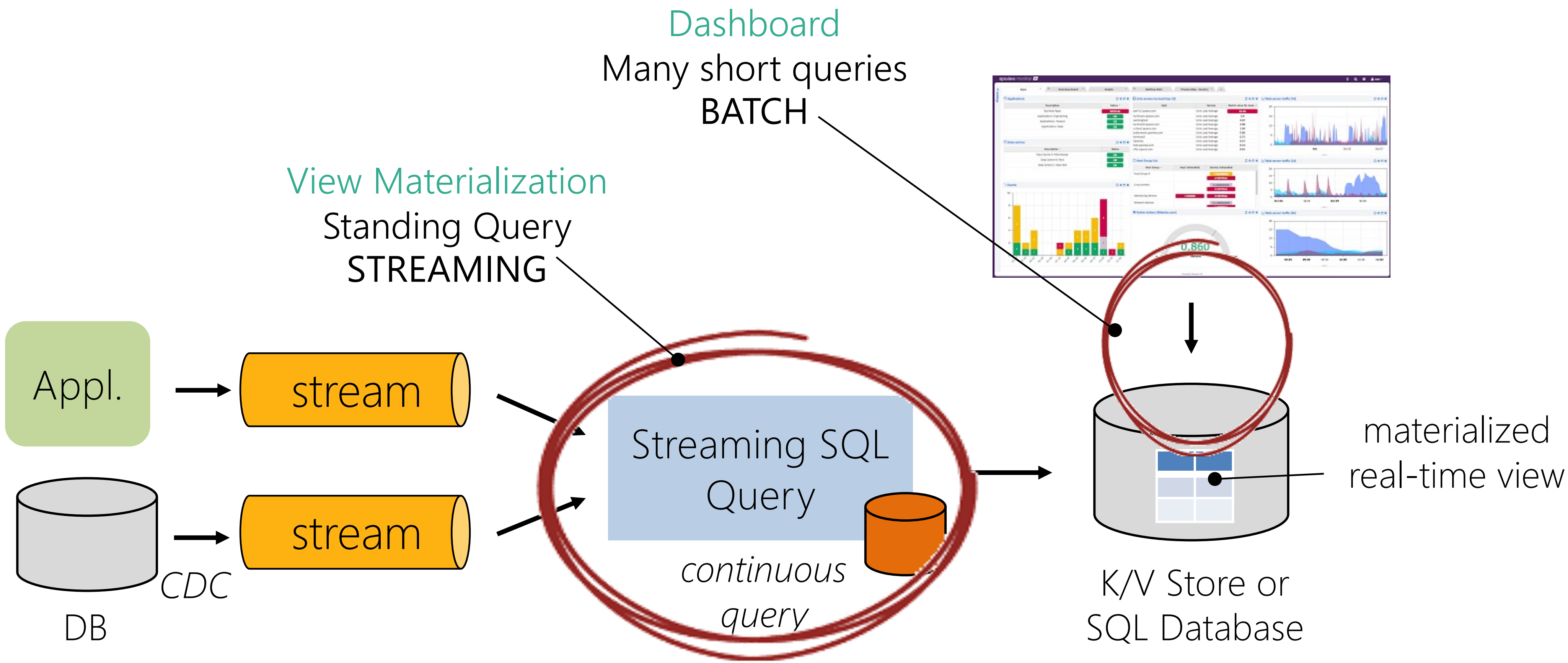
- Support for windowed outer equi-joins

```
SELECT d.rideId, d.departureTime, a.arrivalTime
FROM Departures d LEFT OUTER JOIN Arrivals a
  ON d.rideId = a.rideId
  AND a.arrivalTime BETWEEN
      d.deptureTime AND d.departureTime + '2' HOURS
```

- Support for non-windowed inner joins

```
SELECT u.name, u.address, o.productId, o.amount
FROM Users u JOIN Orders o
  ON u.userId = o.userId
```

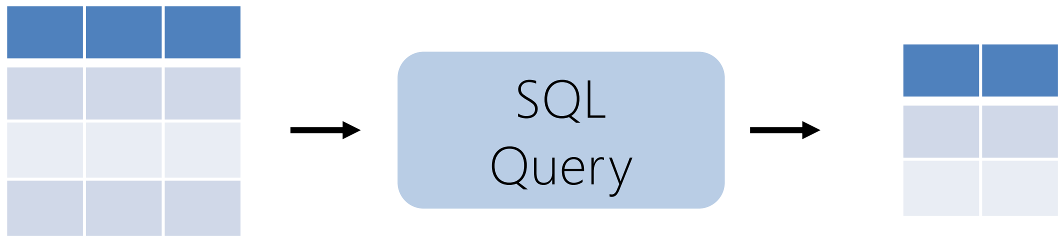
Streaming SQL and Batch SQL



Flink SQL on Data Streams

- Easy, Declarative and concise Relational API
- Tool for a wide range of use cases
- Unification of Batch & Streaming with SAME semantics
- Queries efficiently executed
  - Let Flink handle state, time, and common mistakes

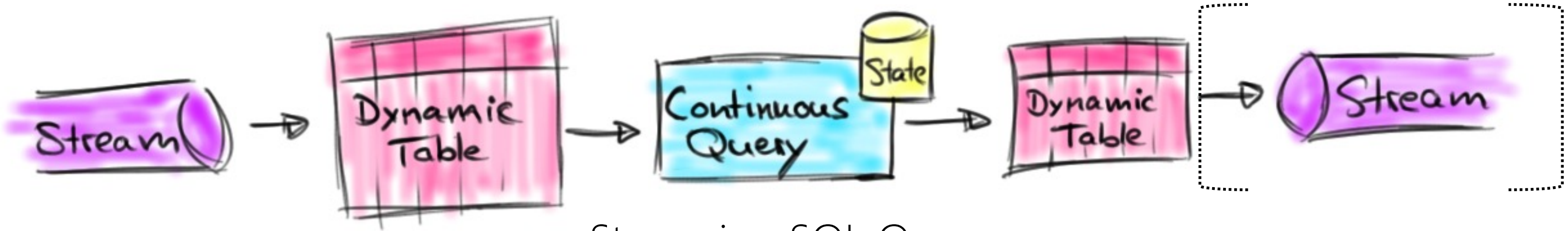
SQL Semantics: Streaming = Batch



input table

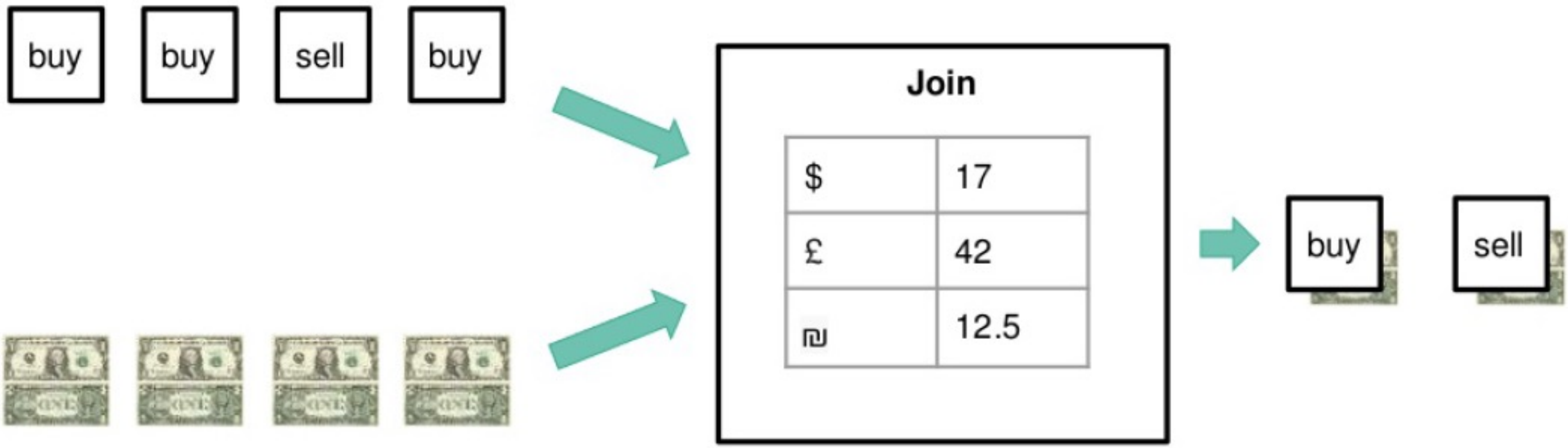
(regular / bounded)
SQL Query

result table

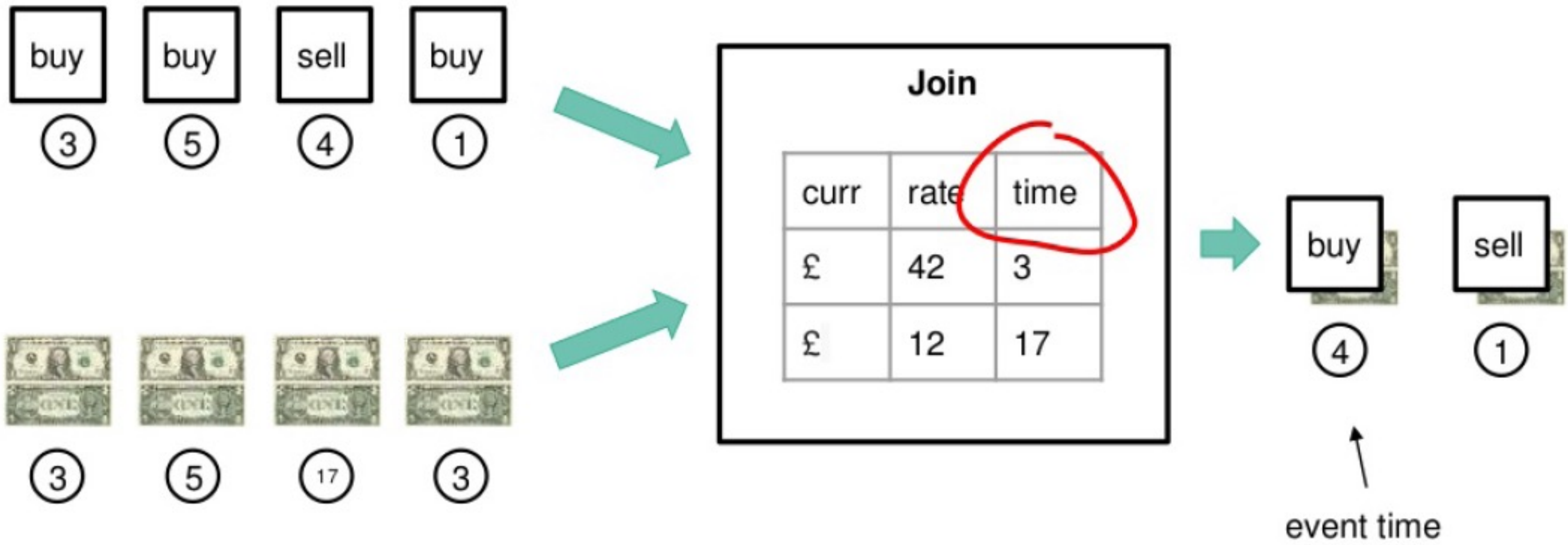


Streaming SQL Query

“Join” me for some trading



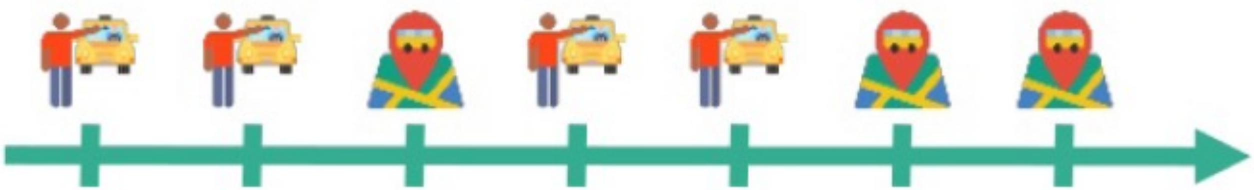
Introducing Time-versioned Table Joins



A Special new Feature for Flink SQL (V1.6 onward)

SQL for pattern analysis?

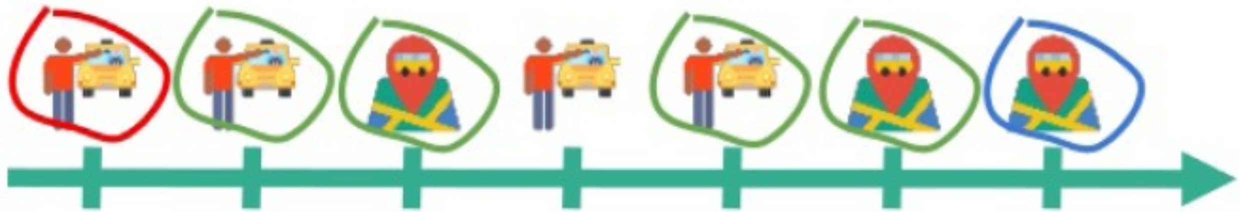
A new Feature for Flink SQL (Beta Rel in V1.7) !



```
SELECT * from ?
```

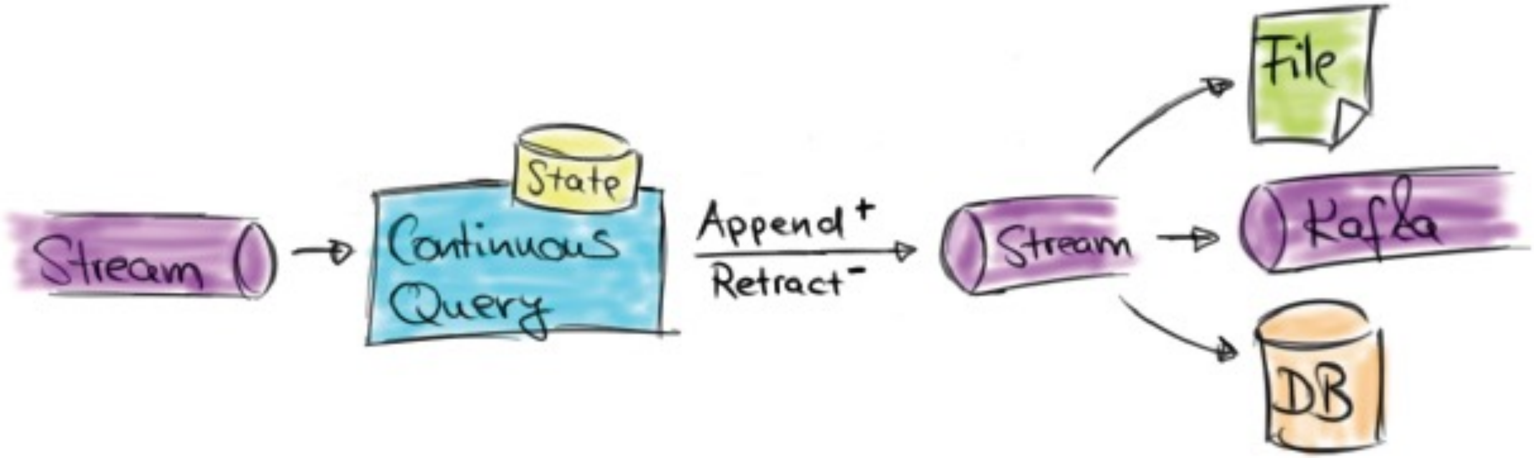
Introducing MATCH_RECOGNIZE

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rideTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S M{2,} E)
    DEFINE
        S AS S.isStart = true,
        M AS M.rideId <> S.rideId,
        E AS E.isStart = false
            AND E.rideId = S.rideId
)
```



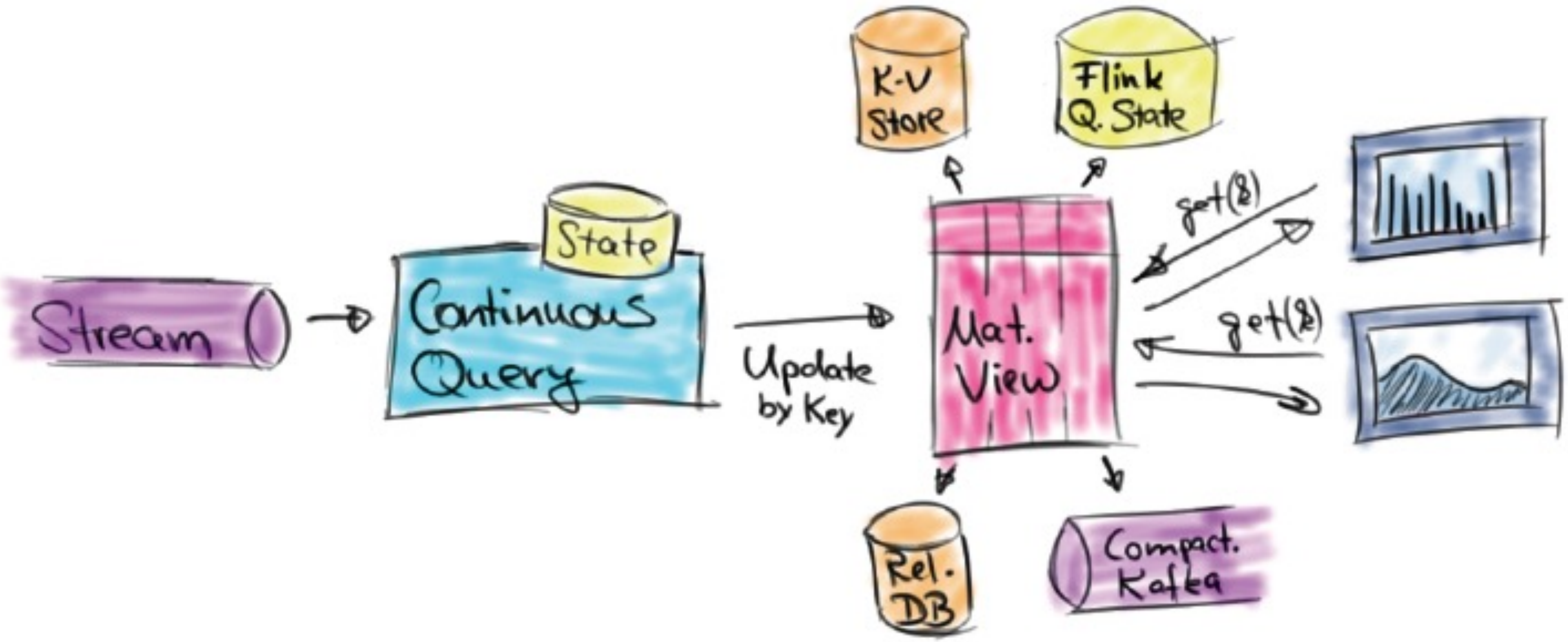
Use Case: Data Pipelines

- Transform, aggregate and move events in real-time
- Low-latency ETL
  - Convert and write streams to file systems, DBMS, K-V stores, indexes, …
  - Ingest appearing files to produce streams



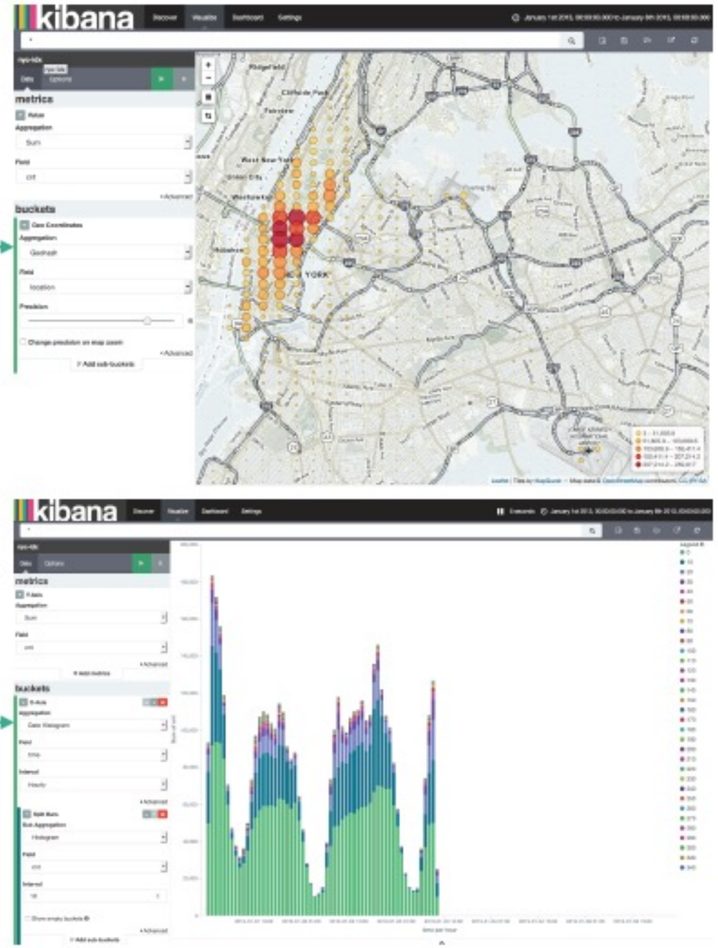
Use Case: Stream & Batch Analytics

- Run analytical queries over bounded and unbounded data
- Query and compare historic and real-time data
- Compute and update data to visualize in real-time



Building a Dashboard Example

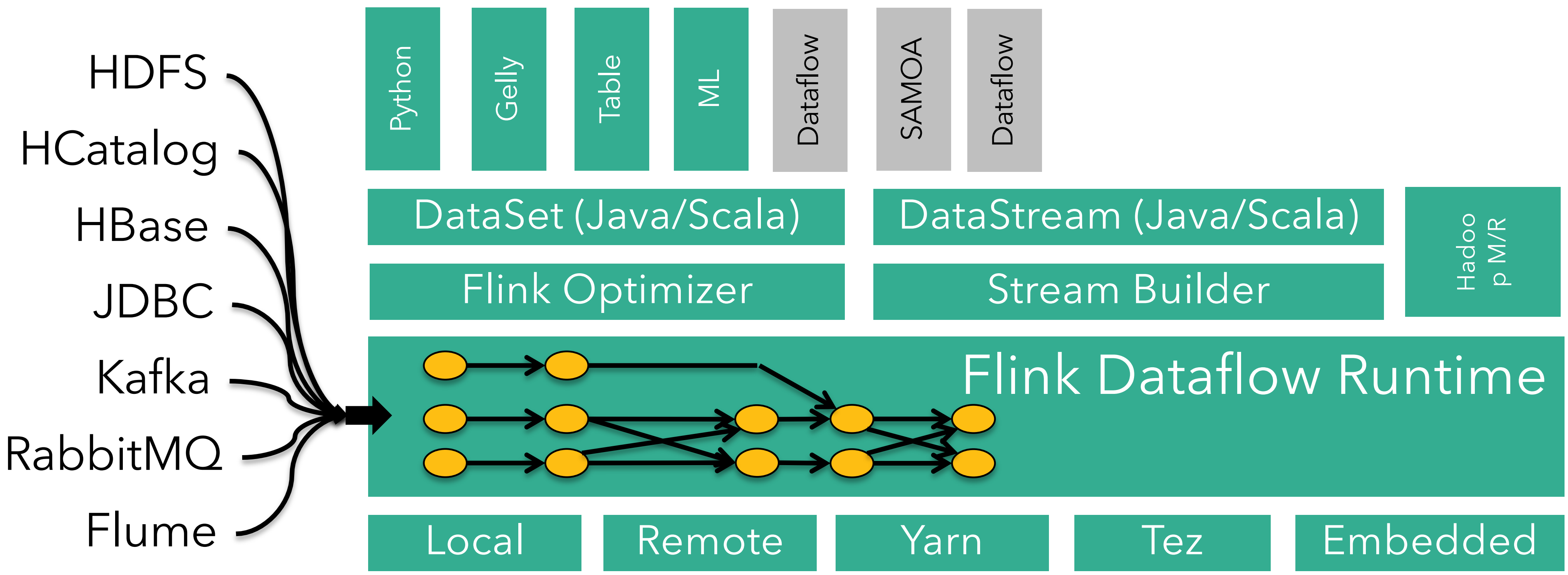
```
SELECT cell,
  isStart,
  HOP_END(rowtime, INTERVAL '5' MINUTE, INTERVAL '15' MINUTE) AS hopEnd,
  COUNT(*) AS cnt
FROM (SELECT rowtime, isStart, toCellId(lon, lat) AS cell
      FROM TaxiRides)
GROUP BY cell,
  isStart,
  HOP(rowtime, INTERVAL '5' MINUTE, INTERVAL '15' MINUTE)
```



Dissecting Flink

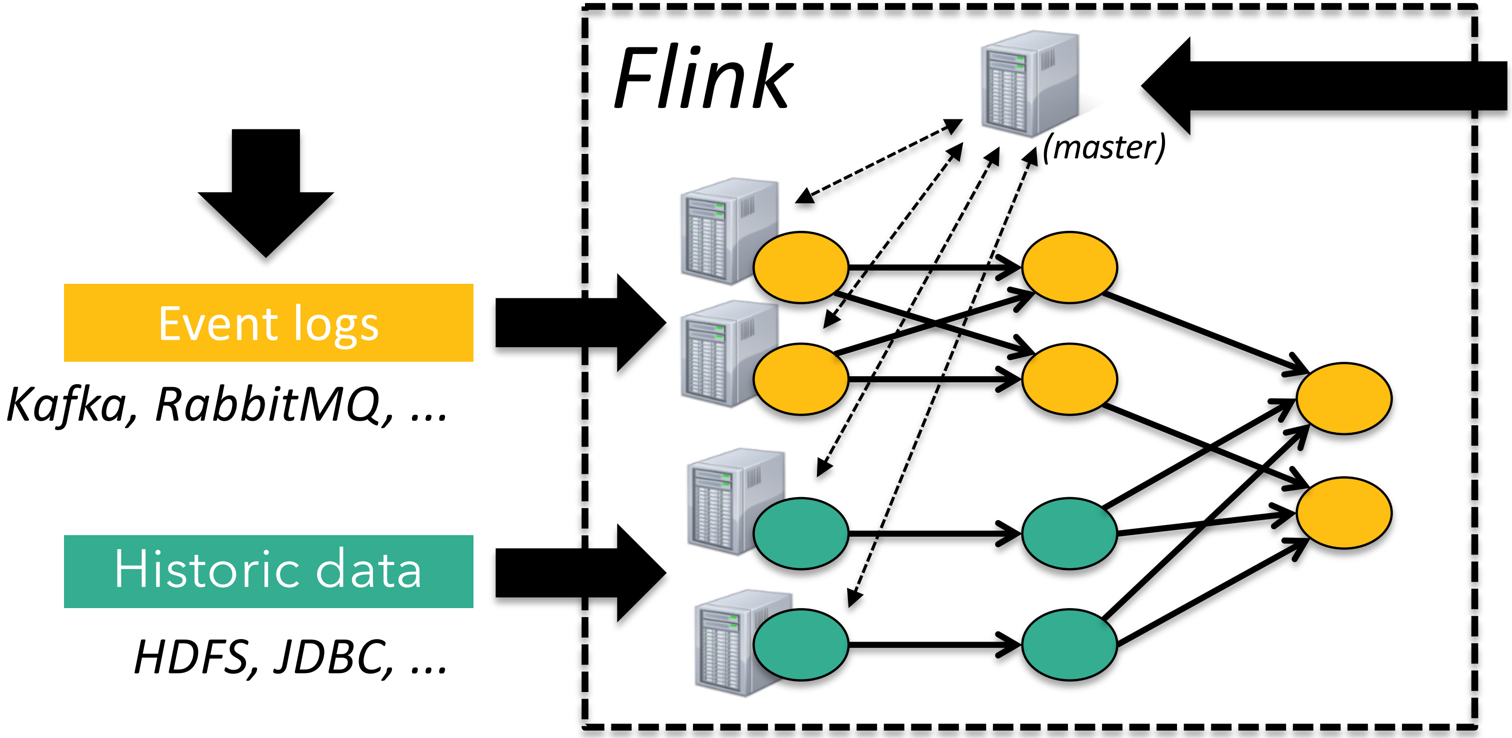
(aka Flink Internals)

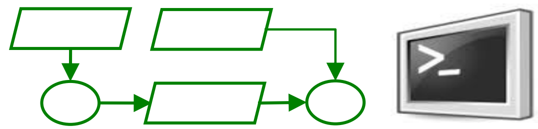
What is Apache Flink?



What is Apache Flink?

Real-time data streams





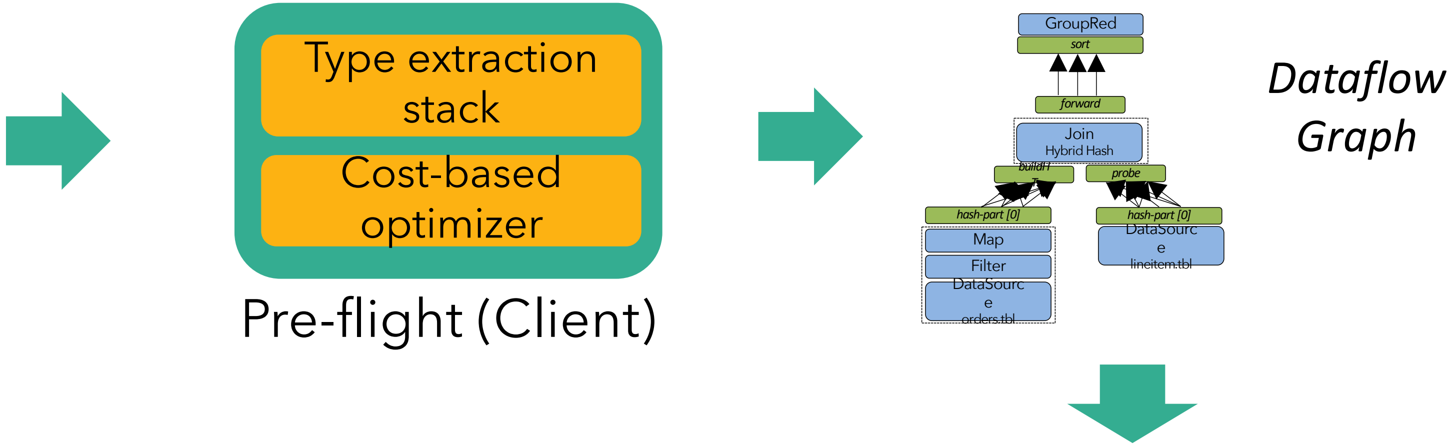
ETL, Graphs, Machine Learning Relational, …

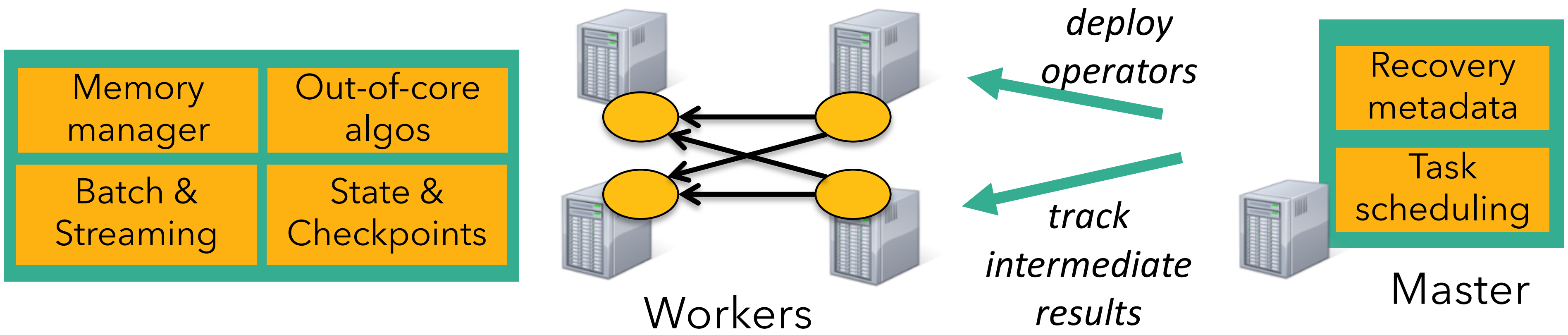
Low latency, windowing, aggregations, ...

Technologies inside Flink

```
case class Path (from: Long, to:
Long)
val tc = edges.iterate(10) {
  paths: DataSet[Path] =>
    val next = paths
      .join(edges)
      .where("to")
      .equalTo("from") {
        (path, edge) =>
          Path(path.from, edge.to)
      }
      .union(paths)
      .distinct()
    next
}
```

Program

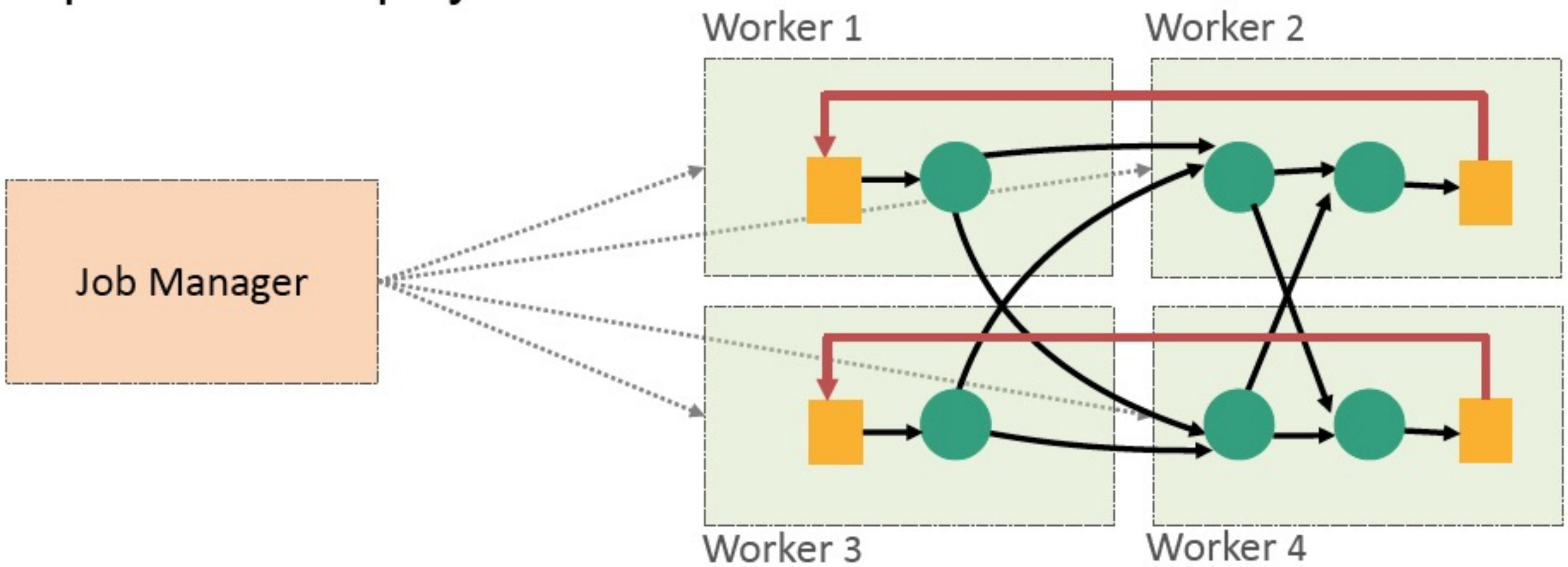




P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas:
Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38(4): 28-38 (2015)

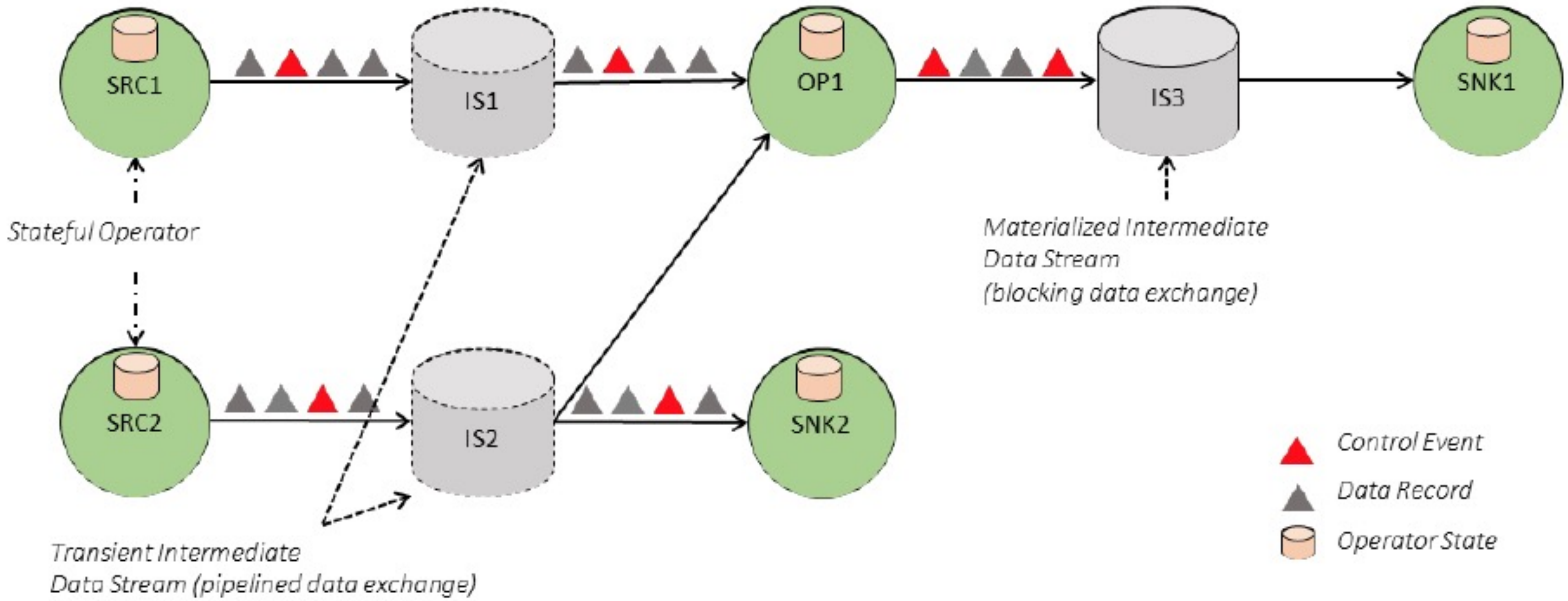
Architecture

- Hybrid MapReduce and MPP database runtime
- Pipelined/Streaming engine
  - Complete DAG deployed



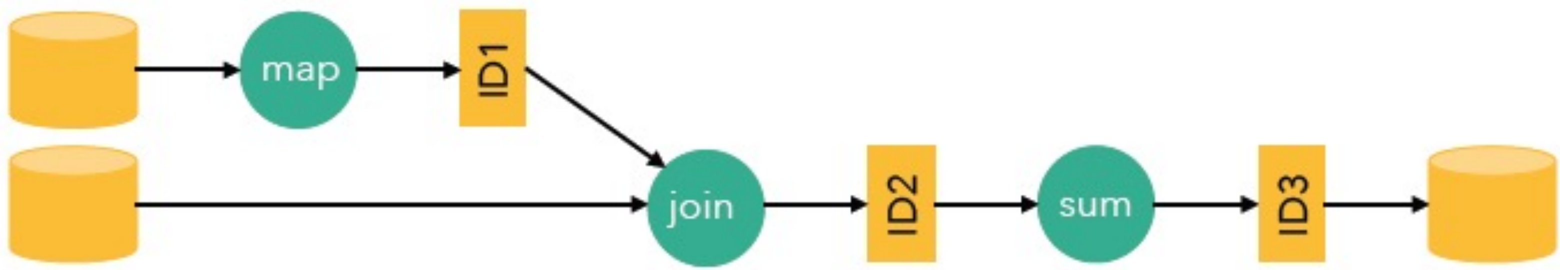
Flink's Pipelined Execution Model

- Flink program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records

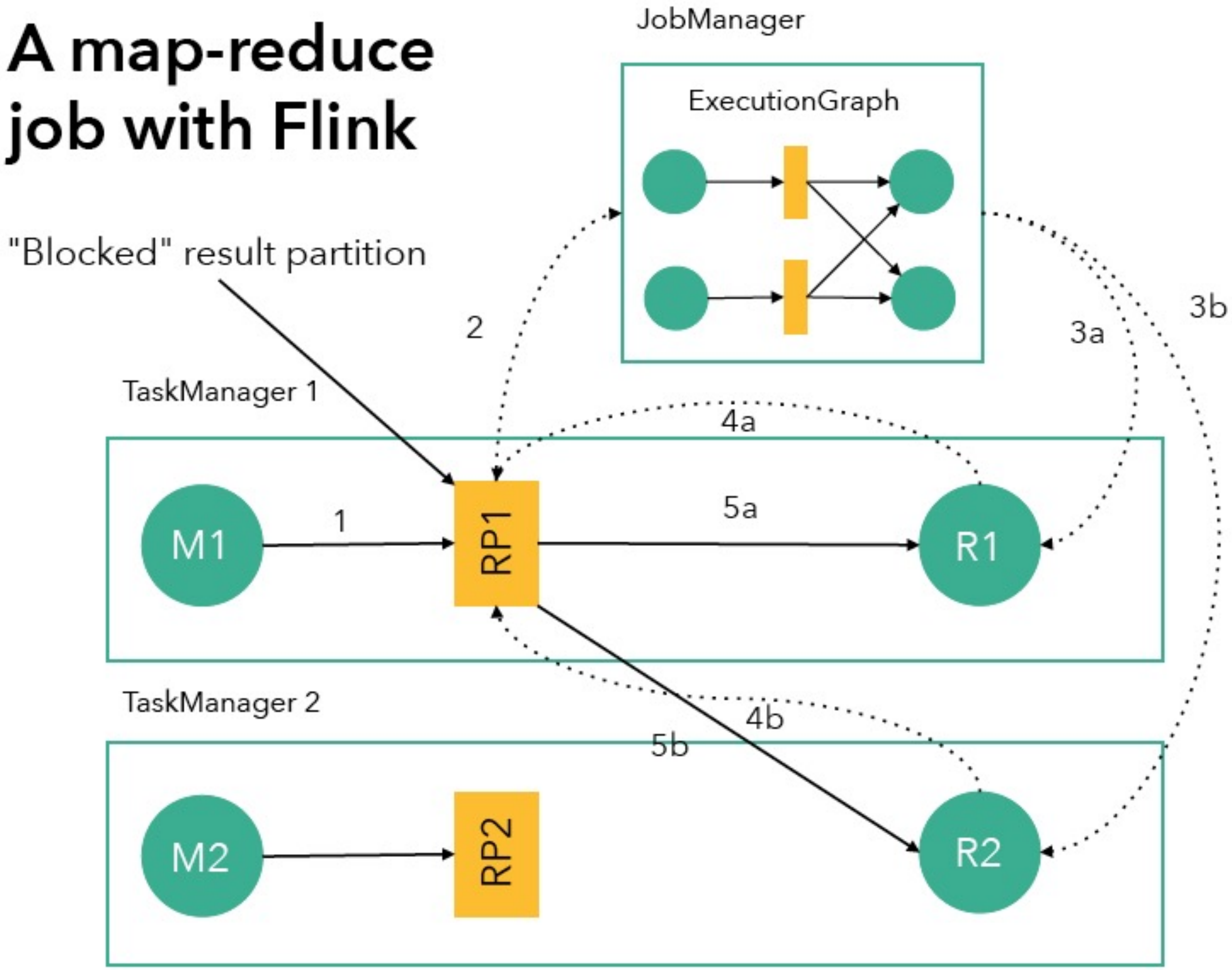


Flink's Execution Model

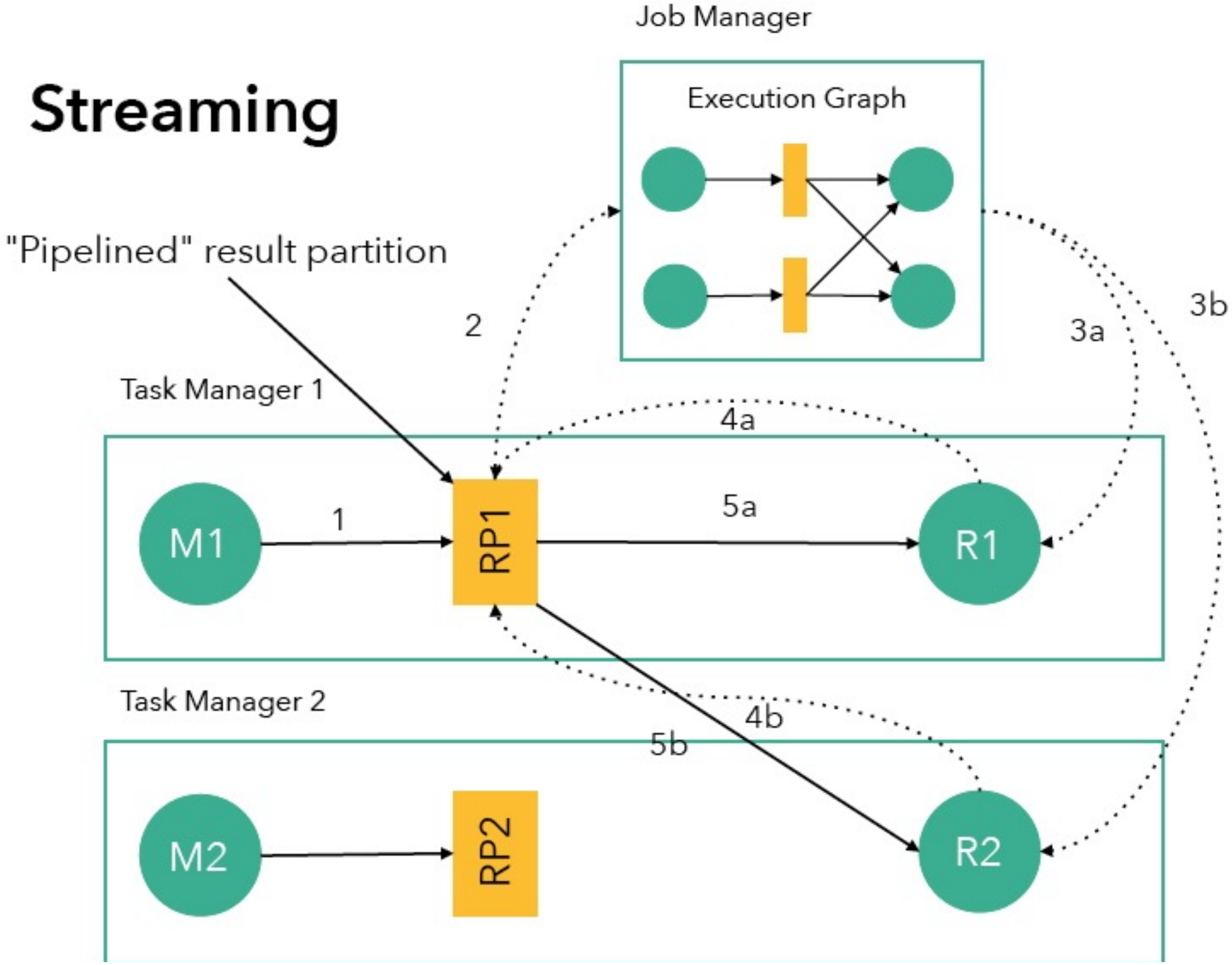
- A program is a graph (DAG) of operators
- Operators = computation + state
- Operators produce intermediate results = logical streams of records
- Other operators can consume those



An Example


A map-reduce job with Flink
JobManager
ExecutionGraph
"Blocked" result partition
2
3a
3b
TaskManager 1
4a
M1
1
RP1
5a
R1
TaskManager 2
4b
5b
M2
RP2
R2

An Example (cont'd)

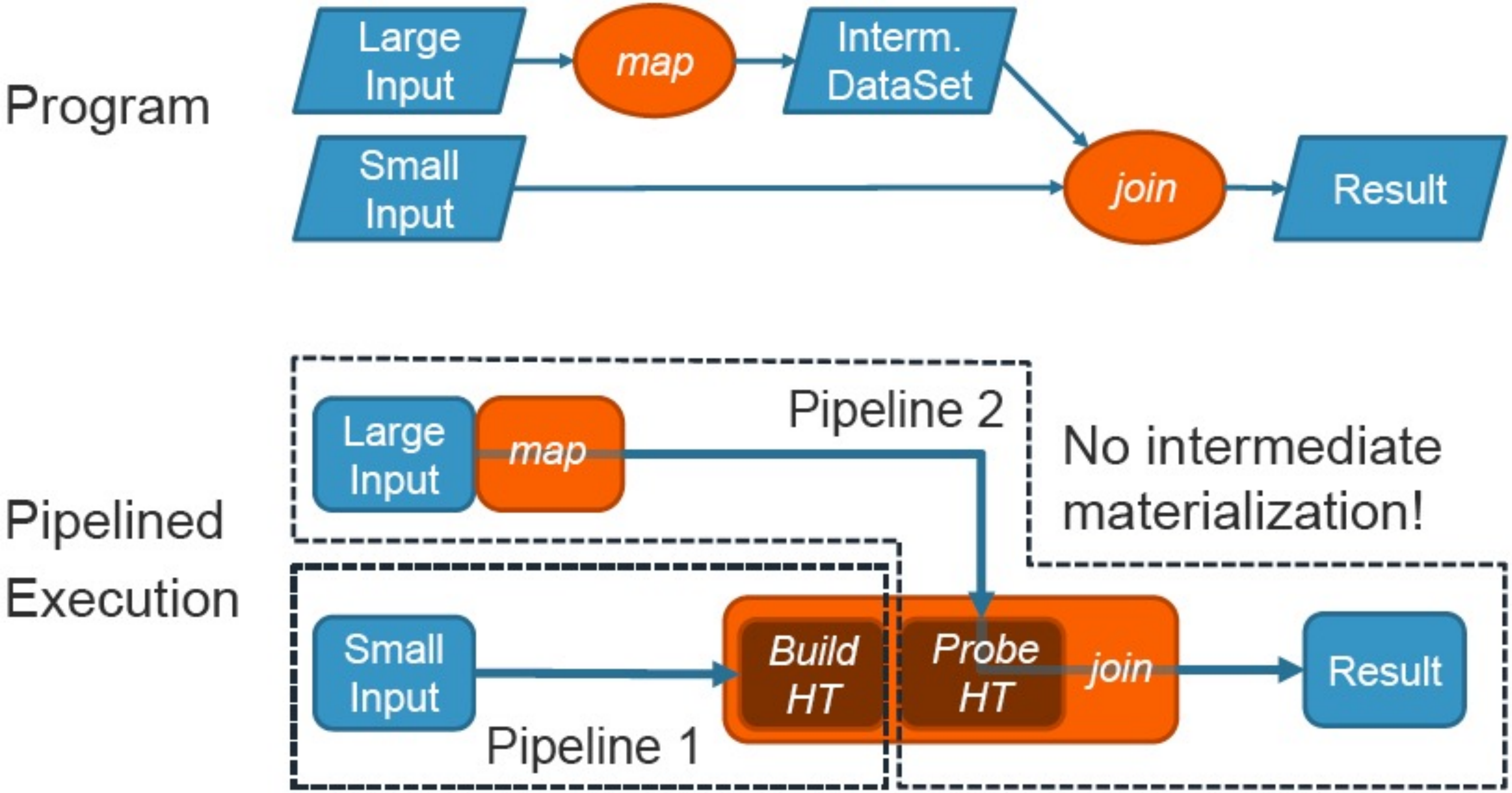


Benefits of Pipelined Data Transfer

- True Stream and Batch Processing in one stack
- Avoid materialization of large intermediate results
- Better performance for many batch workloads

*Flink supports blocking data transfer as well !

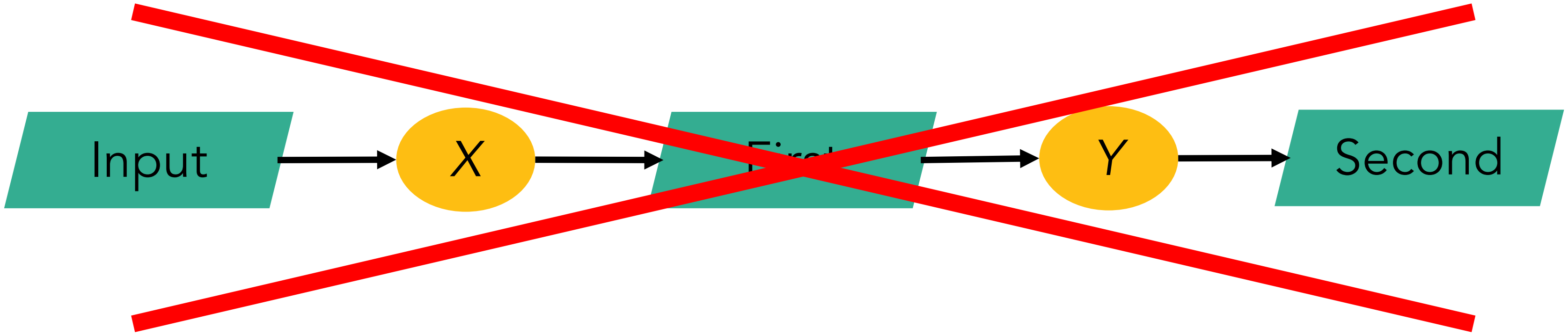
Pipelined Data Transfer



Recap: DataSet

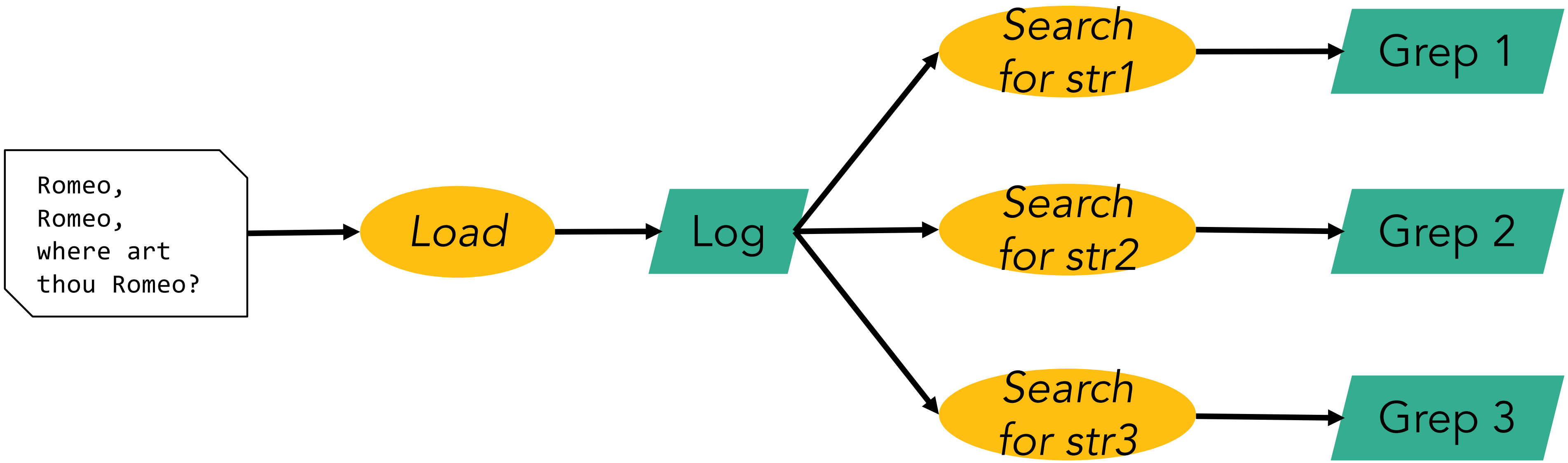

```
ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();
DataSet<String> input = env.readTextFile(input);

DataSet<String> first = input
    .filter (str -> str.contains(“Apache Flink“));
DataSet<String> second = first
    .filter (str -> str.length() > 40);

second.print()
env.execute();
```

Common misconception

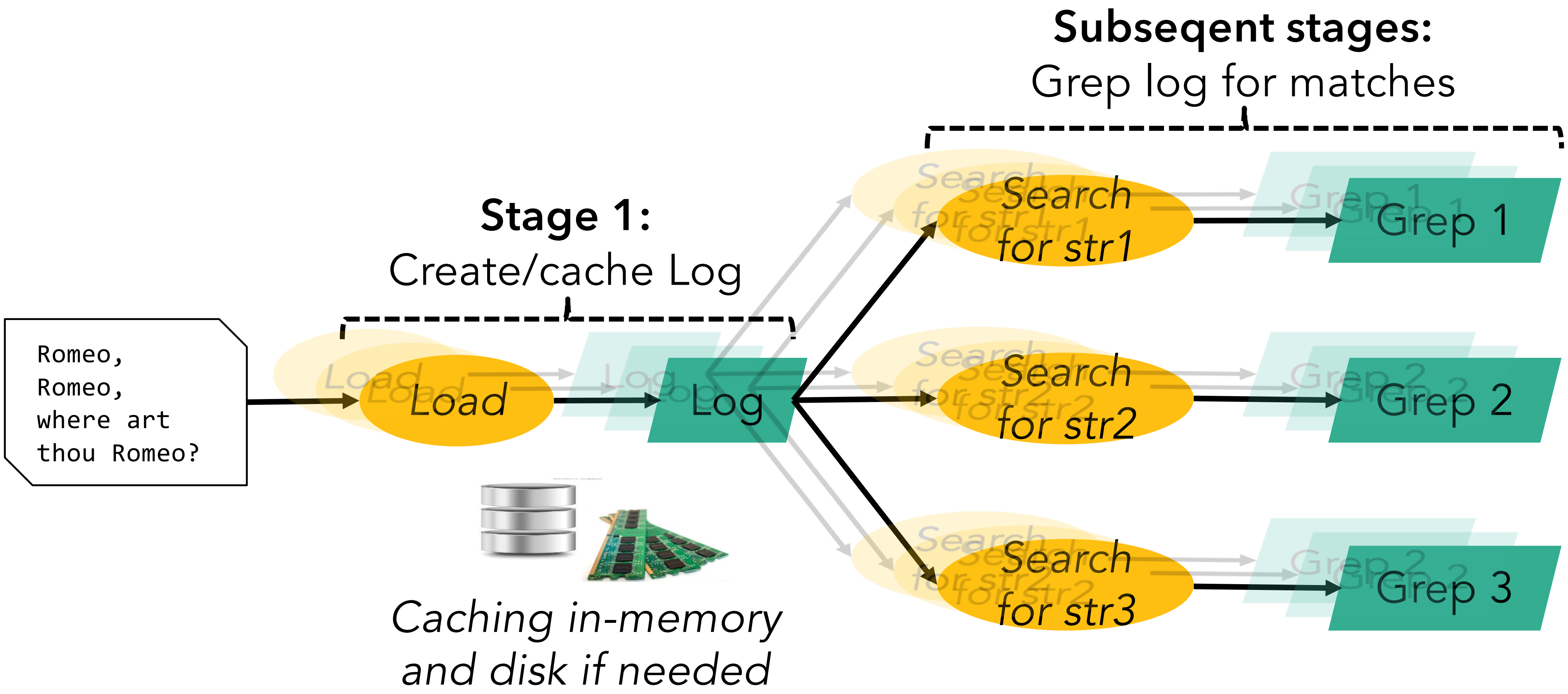


- Programs are not executed eagerly
- Instead, system compiles program to an execution plan and executes that plan

Example: grep



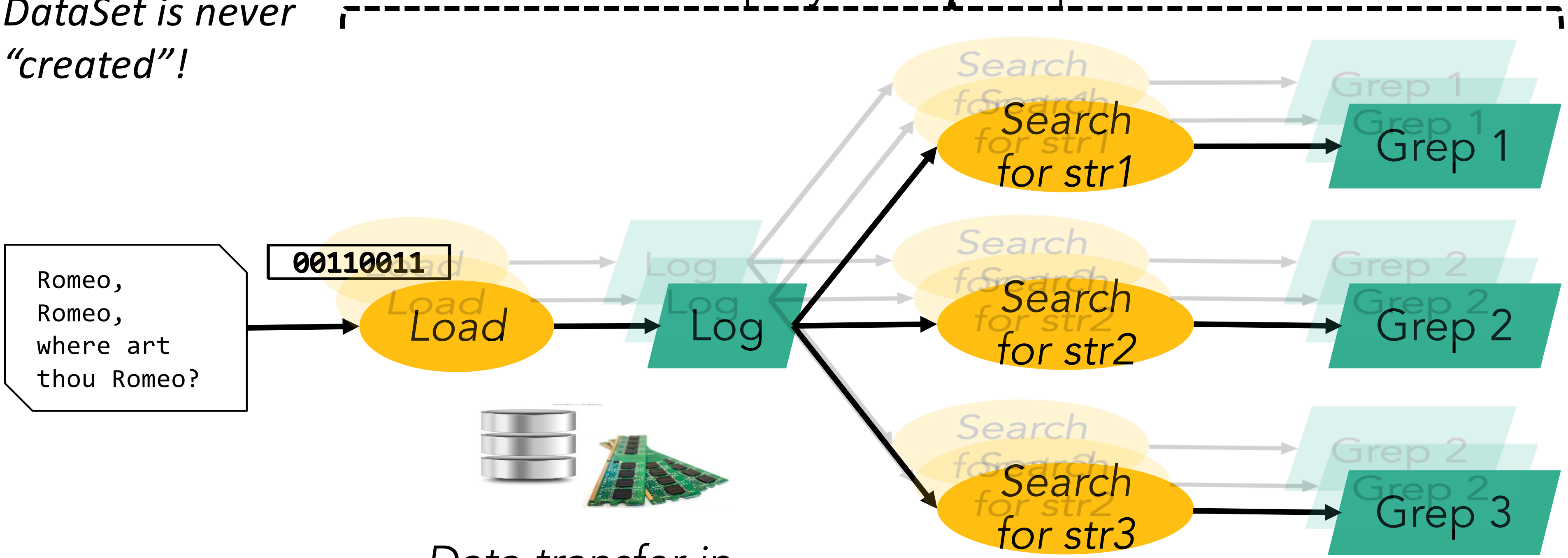
Staged (batch) execution



Pipelined execution

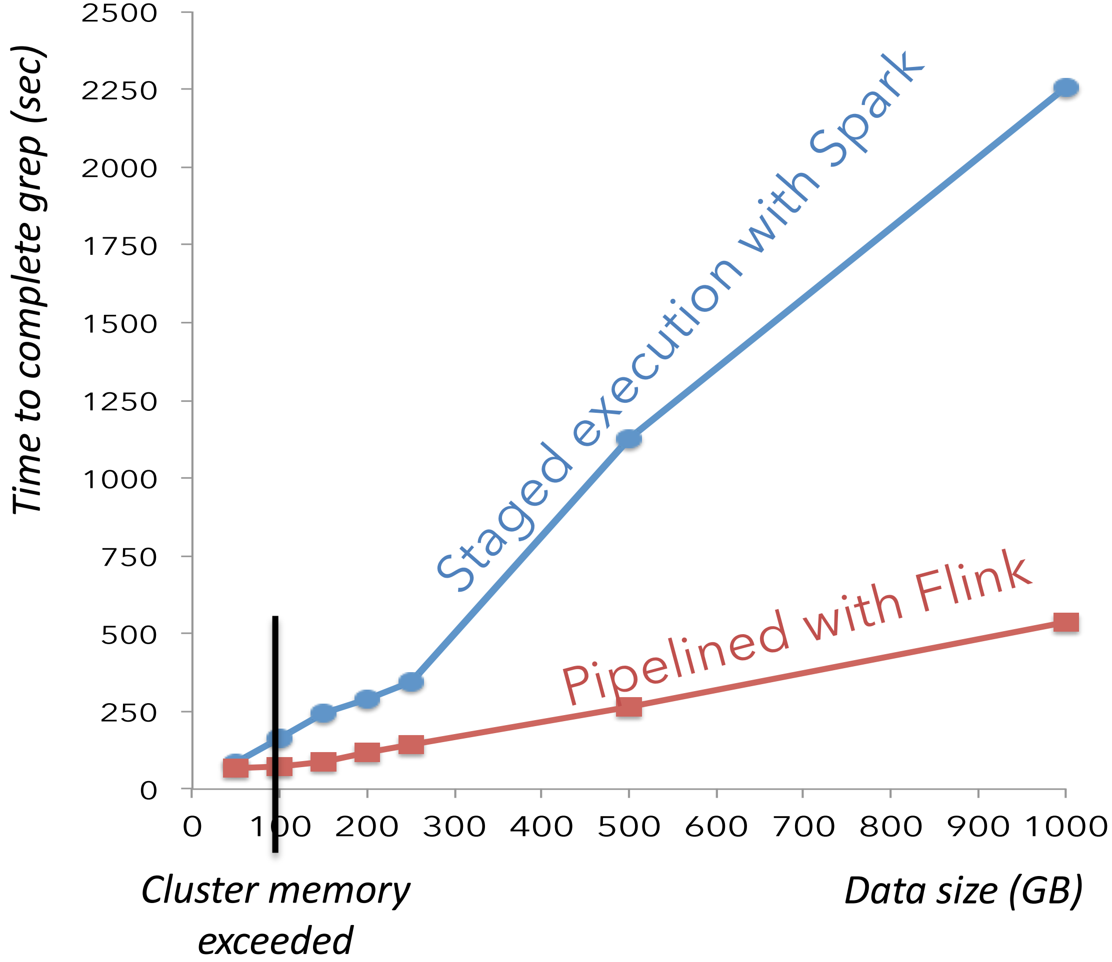
Note: *Log DataSet is never “created”!*

Stage 1:
Deploy and start operators



Data transfer in-memory and disk if needed

Benefits of pipelining



- 25 node cluster
- Grep log for 3 terms
- Scale data size from 100GB to 1TB

▶ Running Jobs

Flink Grep benchmark (10/18/2014, 4:09:39 PM)

cancel

Name	Tasks	Starting	Running	Finished	Canc
DataSource (TextInputFormat (hdfs:/user/robert/datasets/access-1000.log) - UTF-8)	384	0	384	0	0
Filter (grep for lemon)	384	0	384	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_lemon) - UTF-8)	384	49	335	0	0
Filter (grep for tree)	384	0	384	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_tree) - UTF-8)	384	0	384	0	0
Filter (grep for garden)	384	0	384	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_garden) - UTF-8)	384	67	317	0	0
Sum	2688	11	2572	0	0

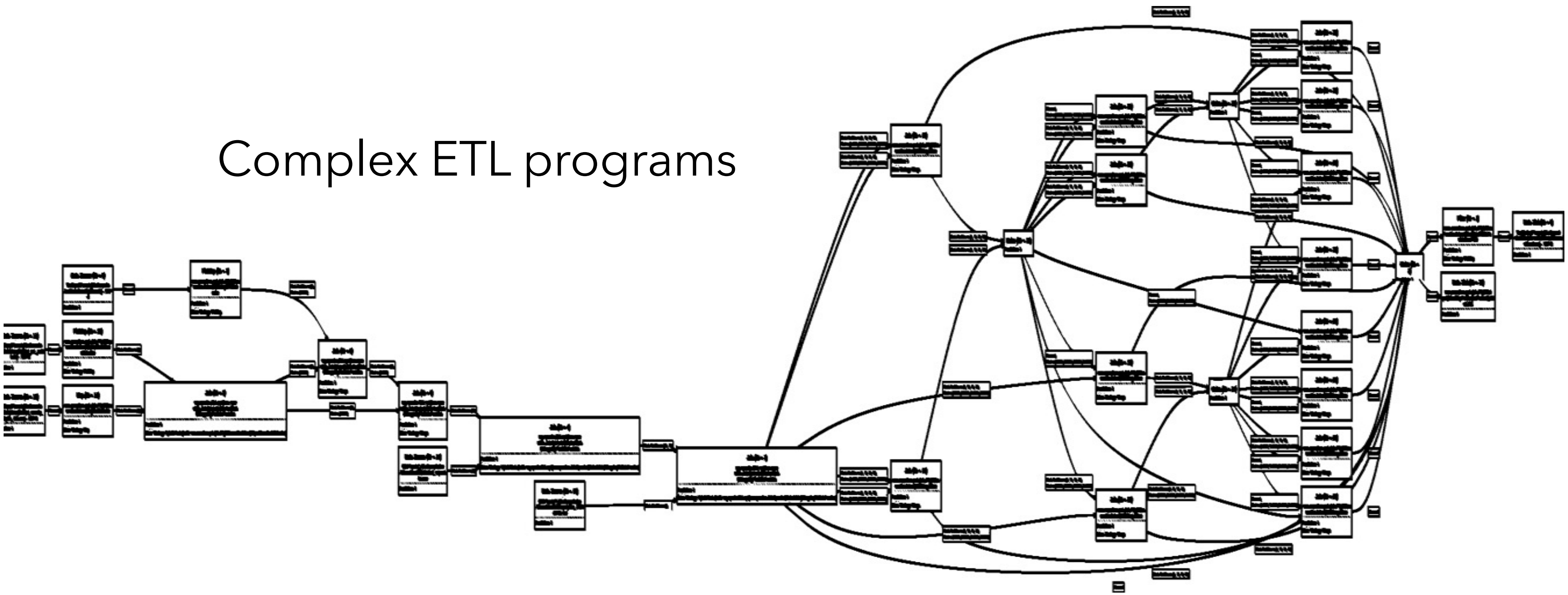
Drawbacks of pipelining

- Long pipelines may be active at the same time leading to memory fragmentation
  - FLINK-1101: Changes memory allocation from static to adaptive

- Fault-tolerance harder to get right
  - FLINK-986: Adds intermediate data sets (similar to RDDS) as first-class citizen to Flink Runtime. Will lead to fine-grained fault-tolerance among other features.

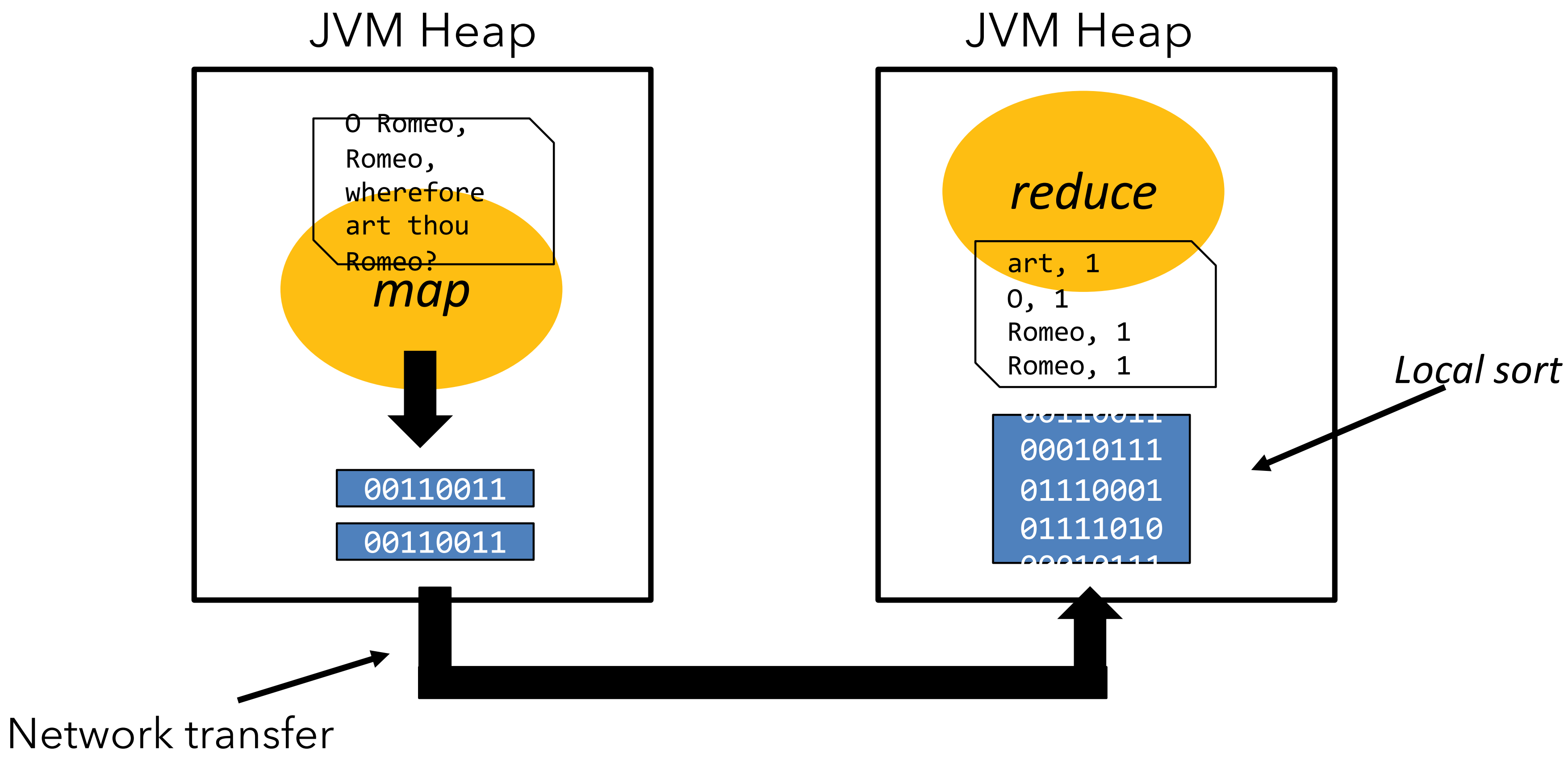
Support Heavy ETL Data Pipelines

Complex ETL programs



Internal data representation

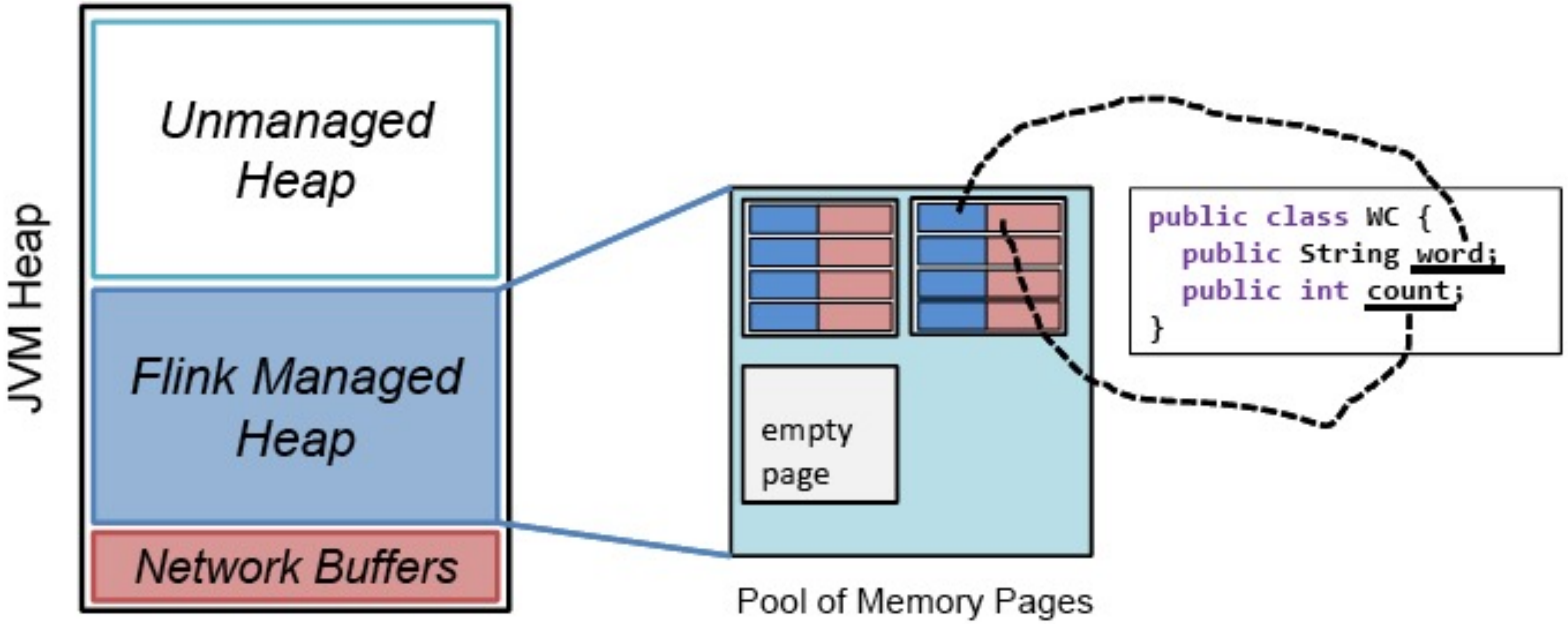
How is intermediate data internally represented?



Internal data representation

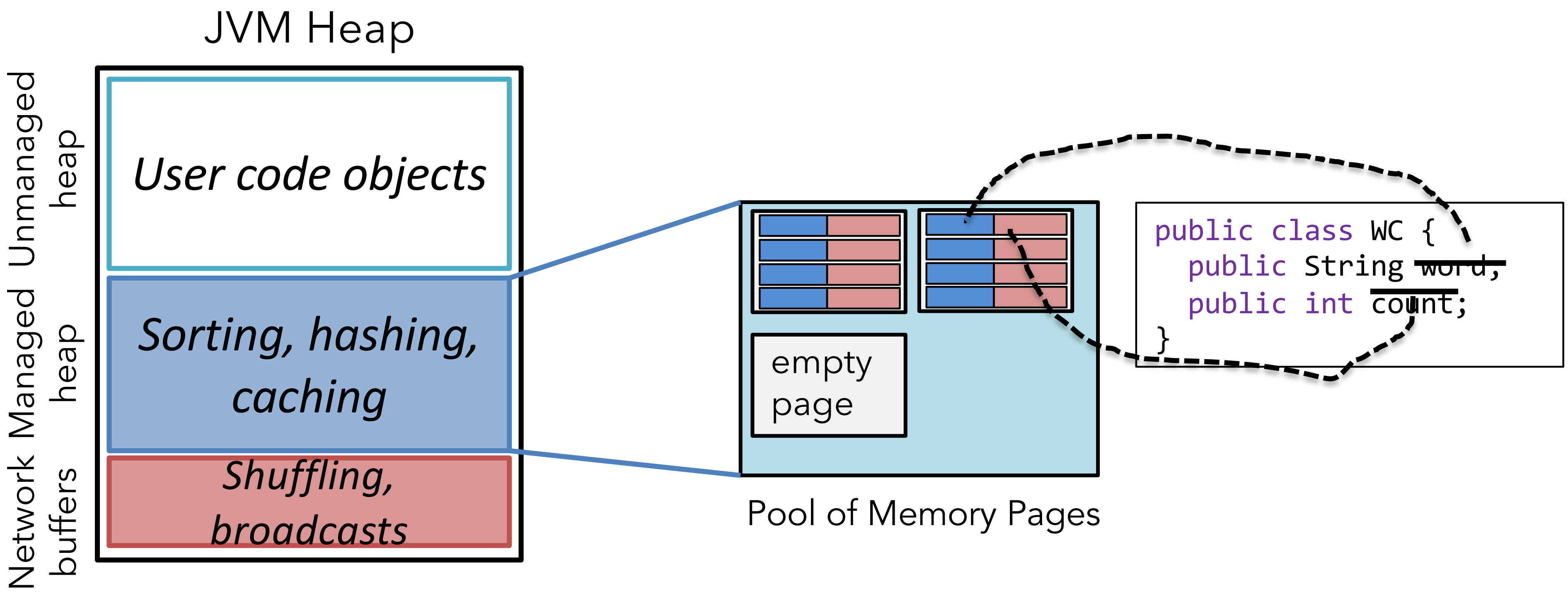
- Two options: Java objects or raw bytes
- **Java objects**
  - Easier to program
  - Can suffer from GC overhead
  - Hard to de-stage data to disk, may suffer from "out of memory exceptions"
- **Raw bytes**
  - Harder to program (customer serialization stack, more involved runtime operators)
  - Solves most of memory and GC problems
  - Overhead from object (de)serialization
- Flink follows the **raw byte** approach

Memory Management in Flink



- Flink manages its own memory
- User data stored in serialized byte arrays
- In-memory caching and data processing happens in a dedicated memory fraction
- Never break the JVM heap
- Very efficient disk spilling and network transfer

Memory in Flink



Memory in Flink (2)

- Internal memory management
  - Flink initially allocates 70% of the free heap as byte[] *segments*
  - Internal operators allocate() and release() these segments

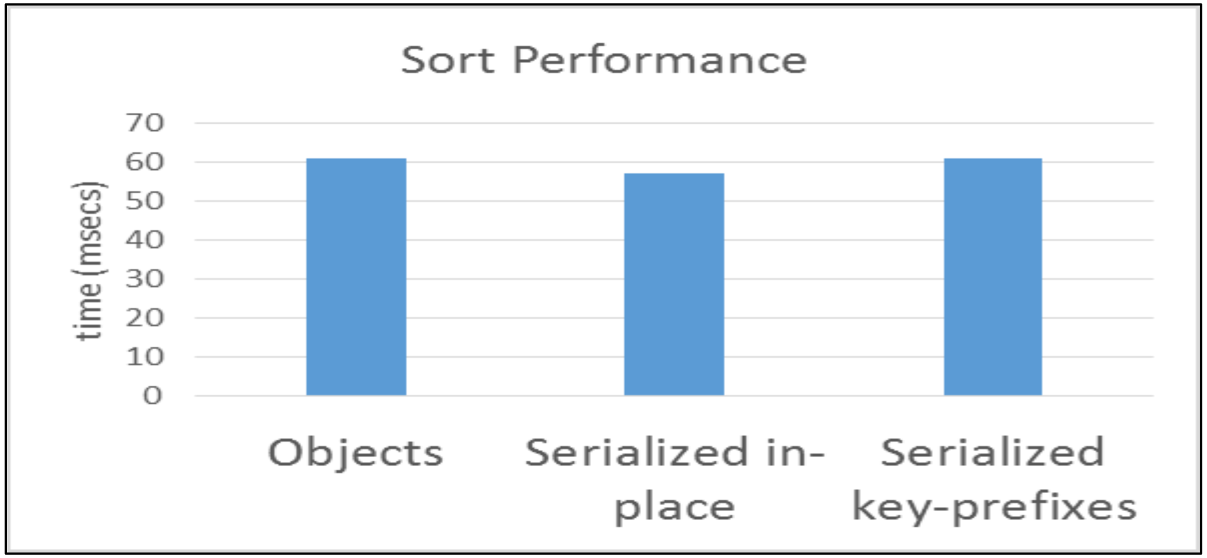
- Flink has its own serialization stack
  - All accepted data types serialized to data segments

- Easy to reason about memory, (almost) no OutOfMemory errors, reduces the pressure to the GC (smooth performance)

Operating on serialized data

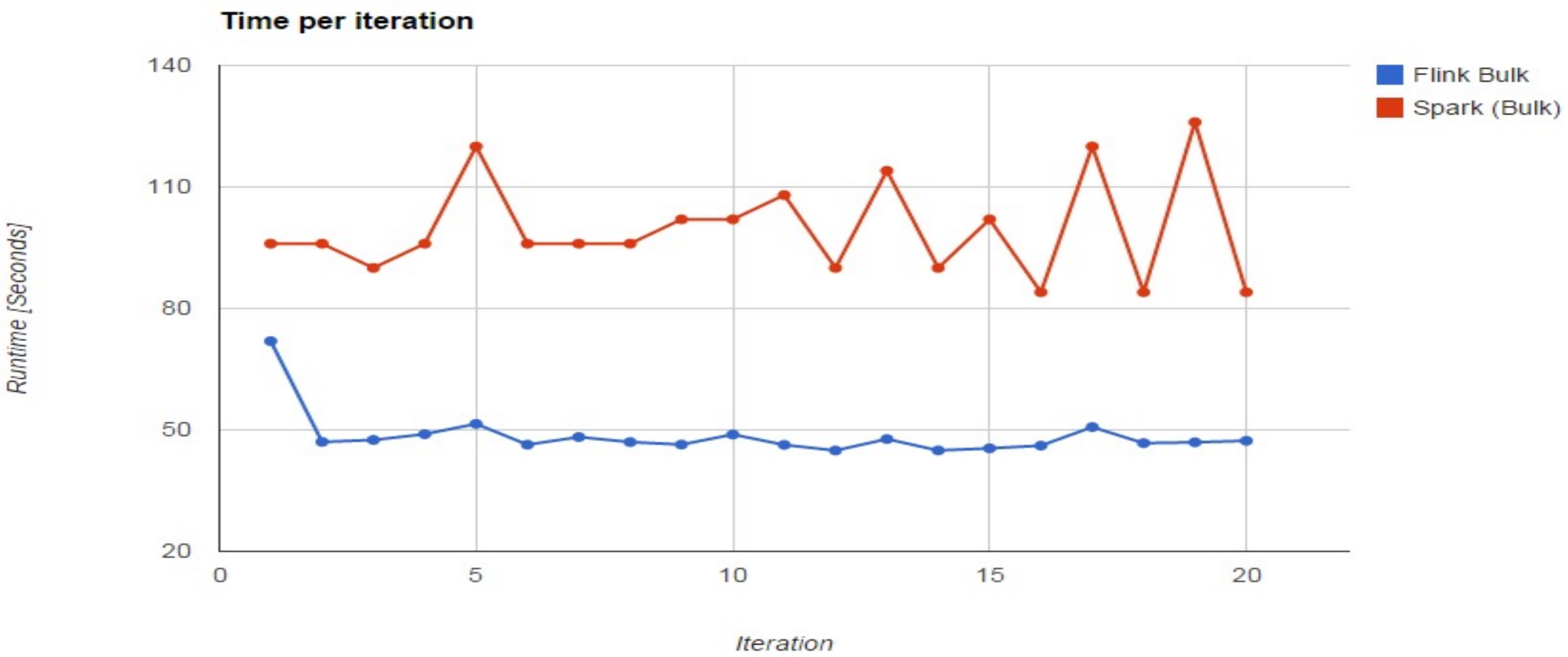
Microbenchmark

- Sorting 1GB worth of (long, double) tuples
- 67,108,864 elements
- Simple quicksort

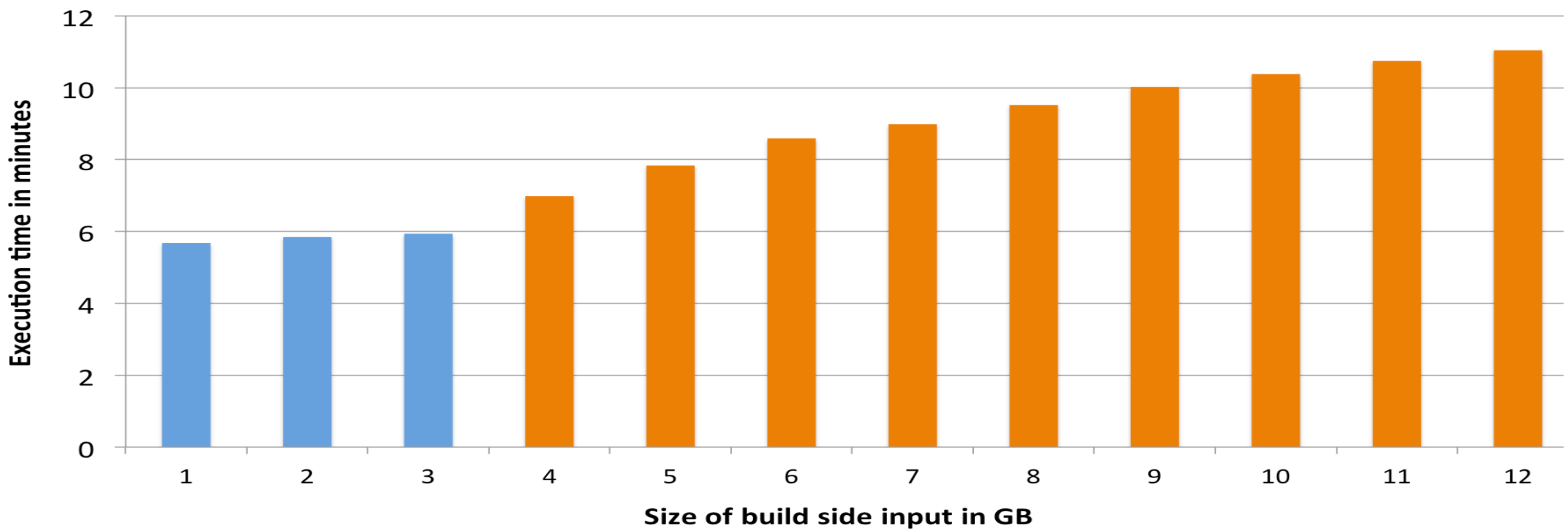


Benefits of managed memory

- More reliable and stable performance (less GC effects, easy to go to disk)



Smooth out-of-core performance

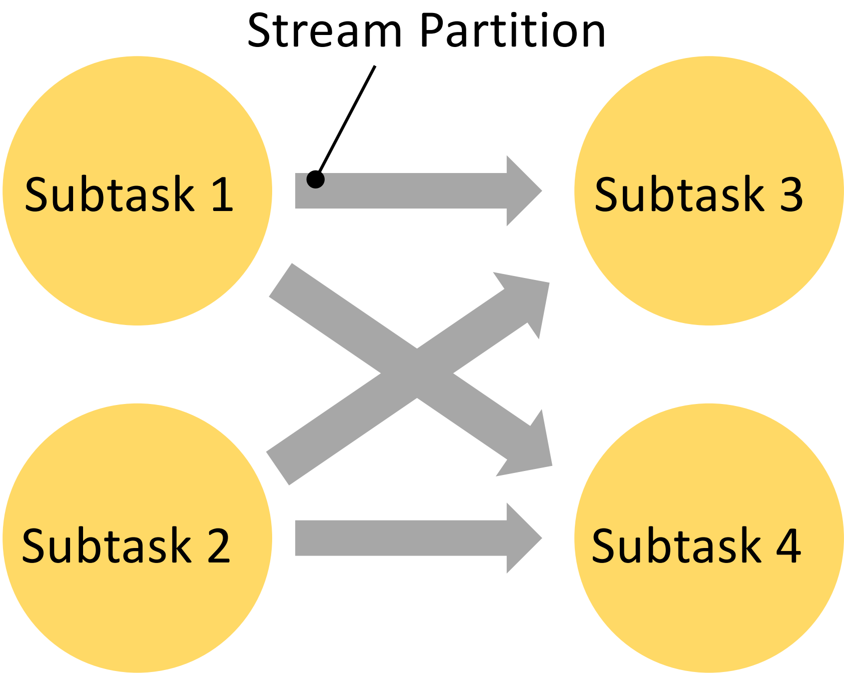


Single-core join of 1KB Java objects beyond memory (4 GB)
Blue bars are in-memory, orange bars (partially) out-of-core

More at: http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html

Network Stack

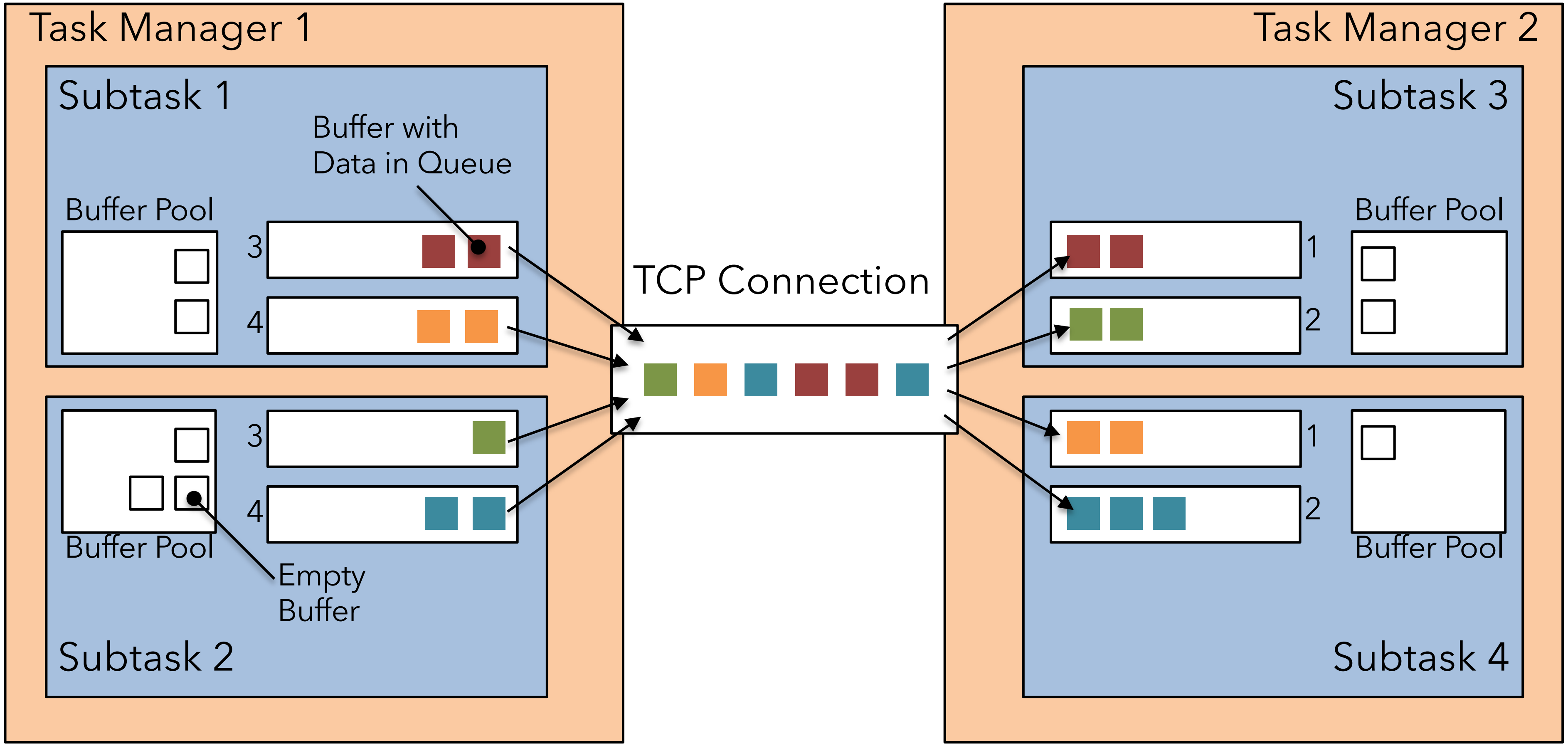
Flink Data Transport (logical)



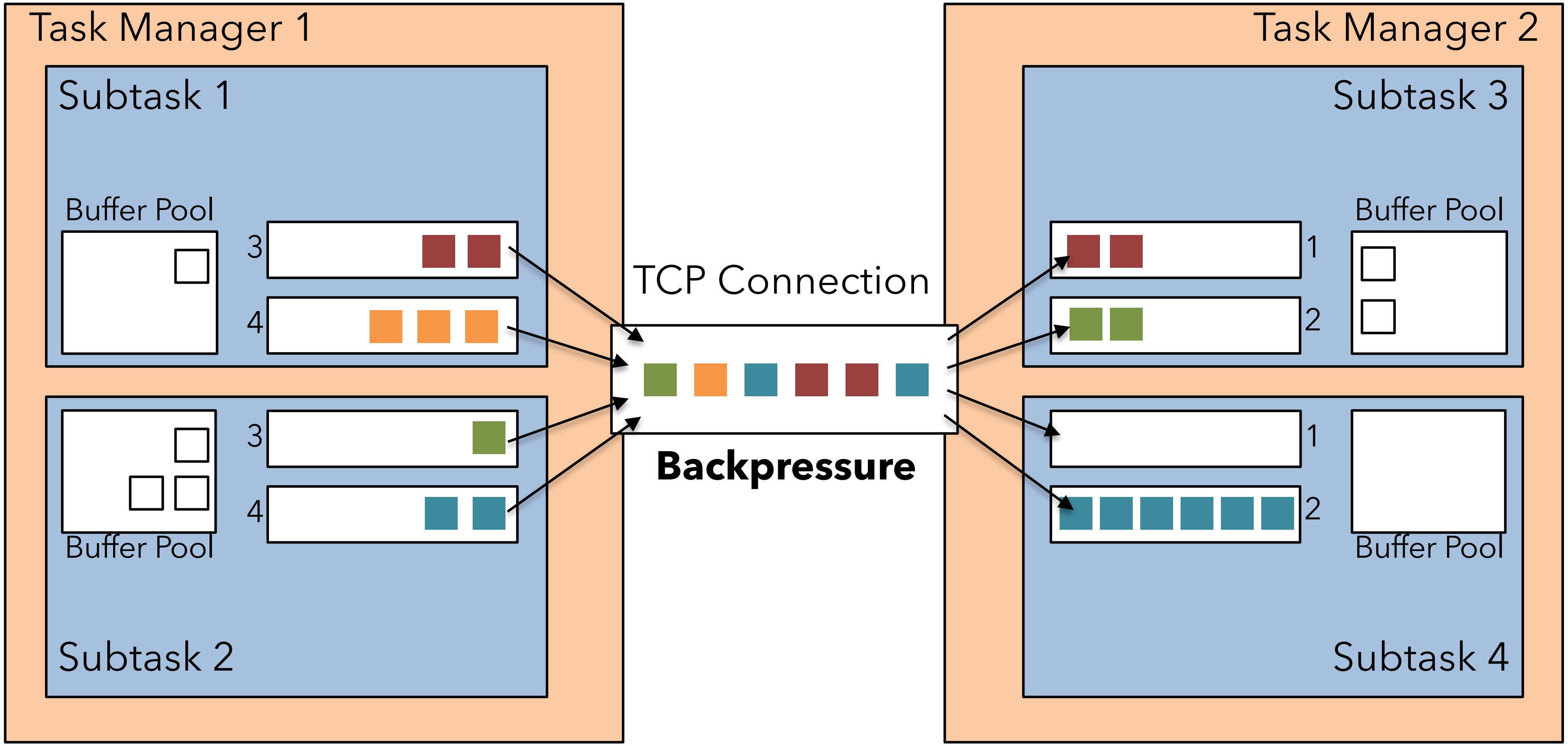
Abstraction over:

- Subtask output
  - pipelined-bounded
  - pipelined-unbounded
  - Blocking
- Scheduling type
  - all at once
  - next stage on complete output
  - next stage on first output
- Transport
  - high throughput via buffers
  - low latency via buffer timeout

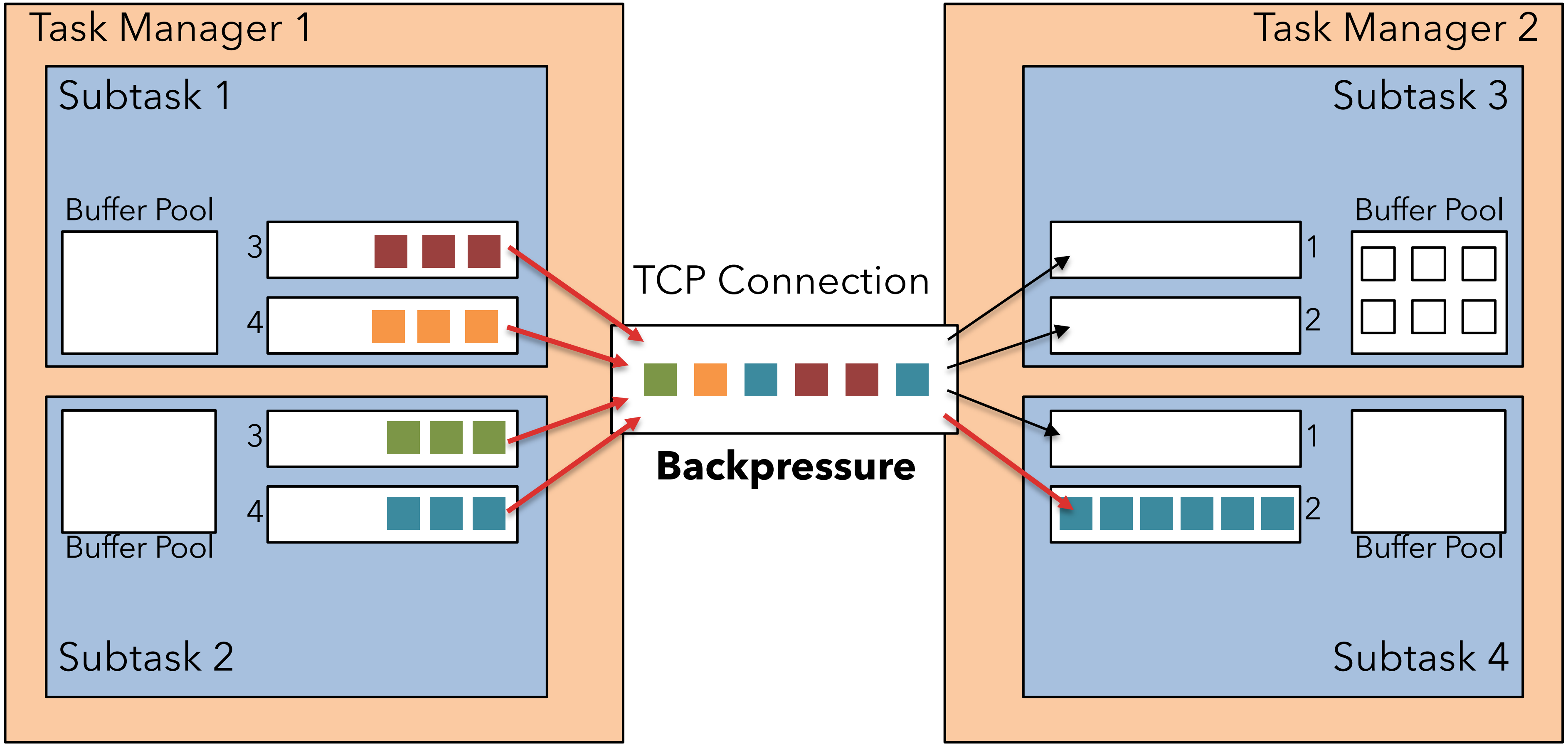
Flink Data Transport (physical)



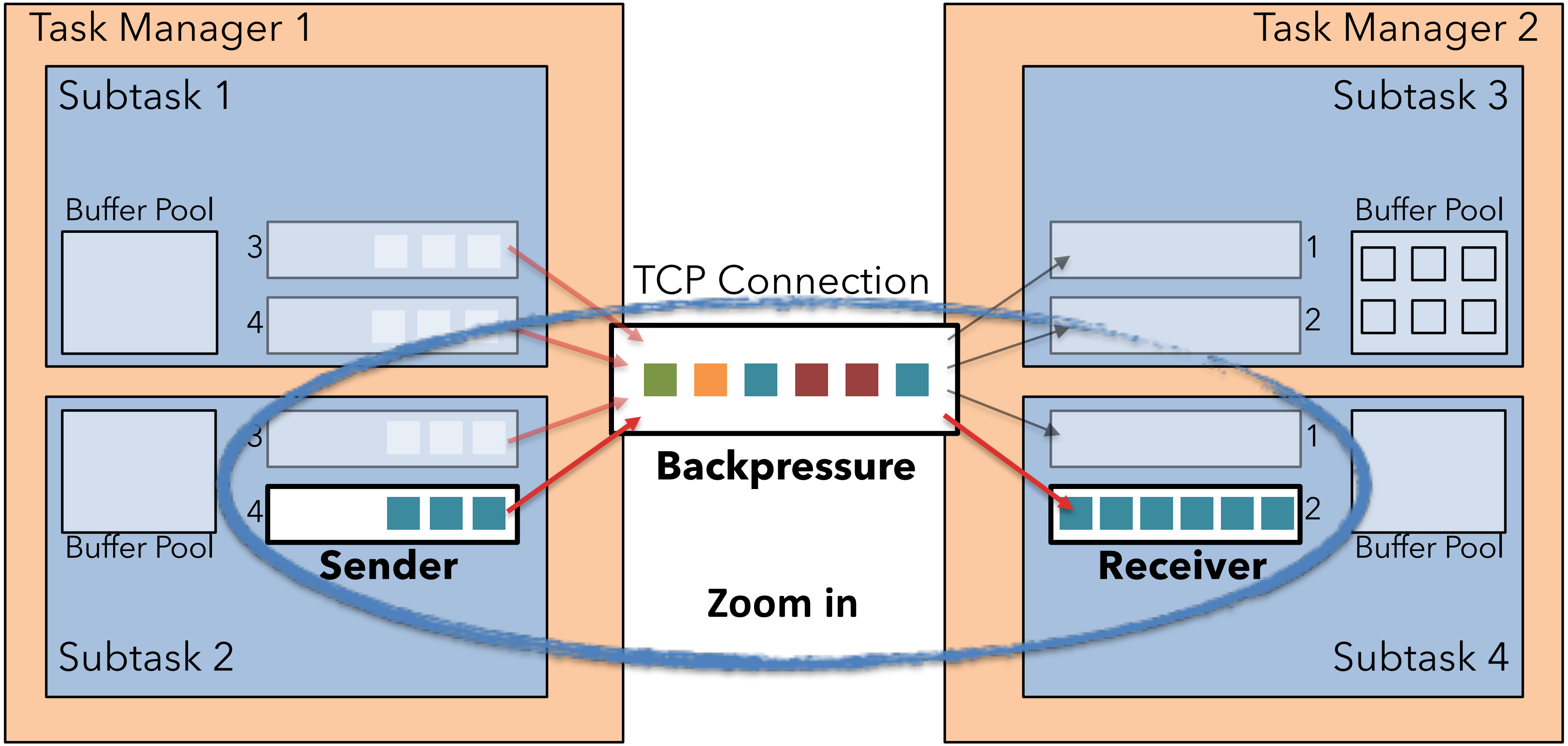
Flink Data Transport (physical)



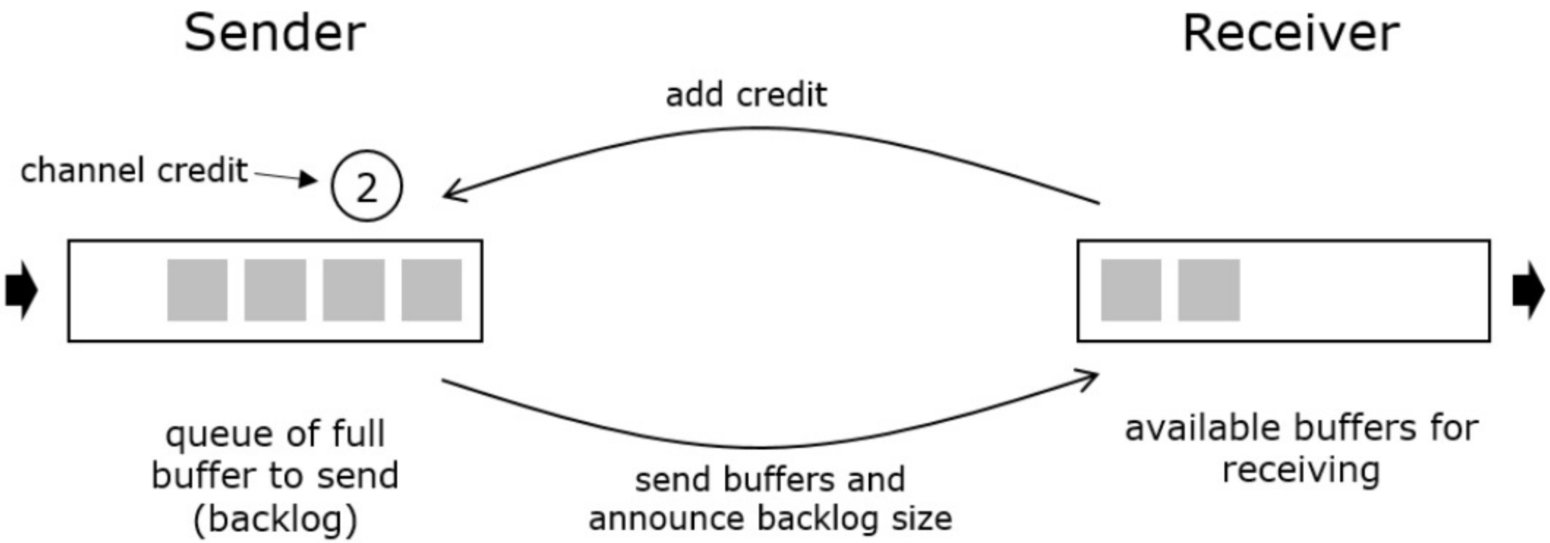
Flink Data Transport (physical)



Flink Data Transport (physical)



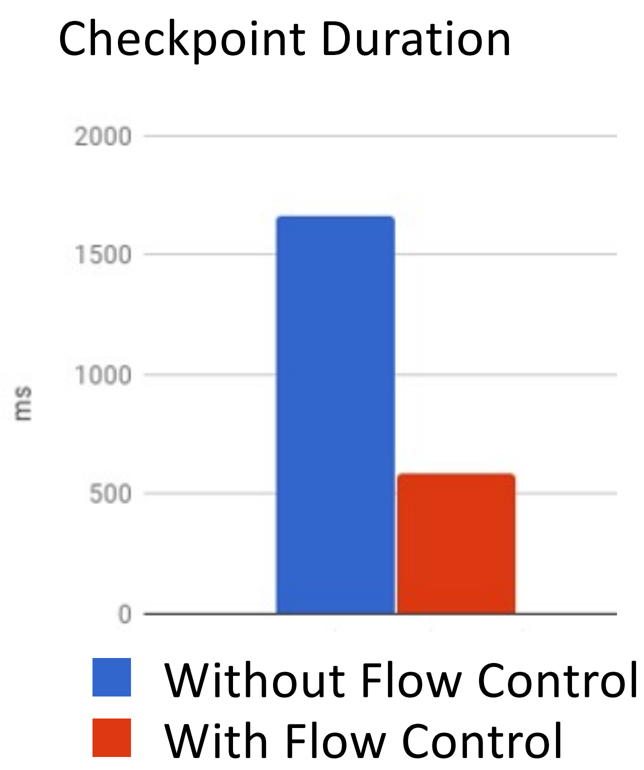
Credit-based Flow Control (Flink 1.5)



- Sender announces backlog.
- Receiver attempts to allocate buffers.
- Receiver gives credit for allocated buffers.
- Result: Never blocks on the TCP connection.

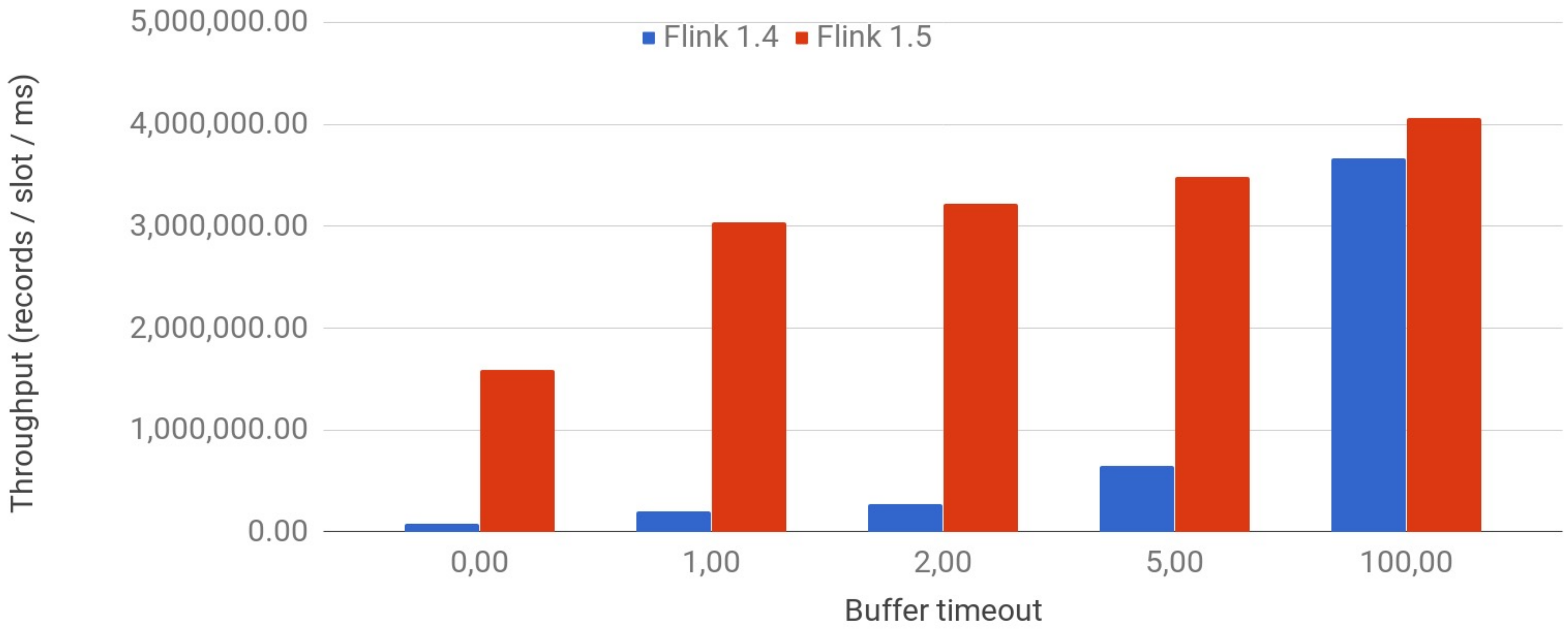
Credit-based Flow Control (Flink 1.5)

- Never blocks the TCP connection
- Avoids overloading of slow receivers
- Improves checkpoint alignment



Reduced Overhead

- low latency via buffer timeout
- high throughput through buffers

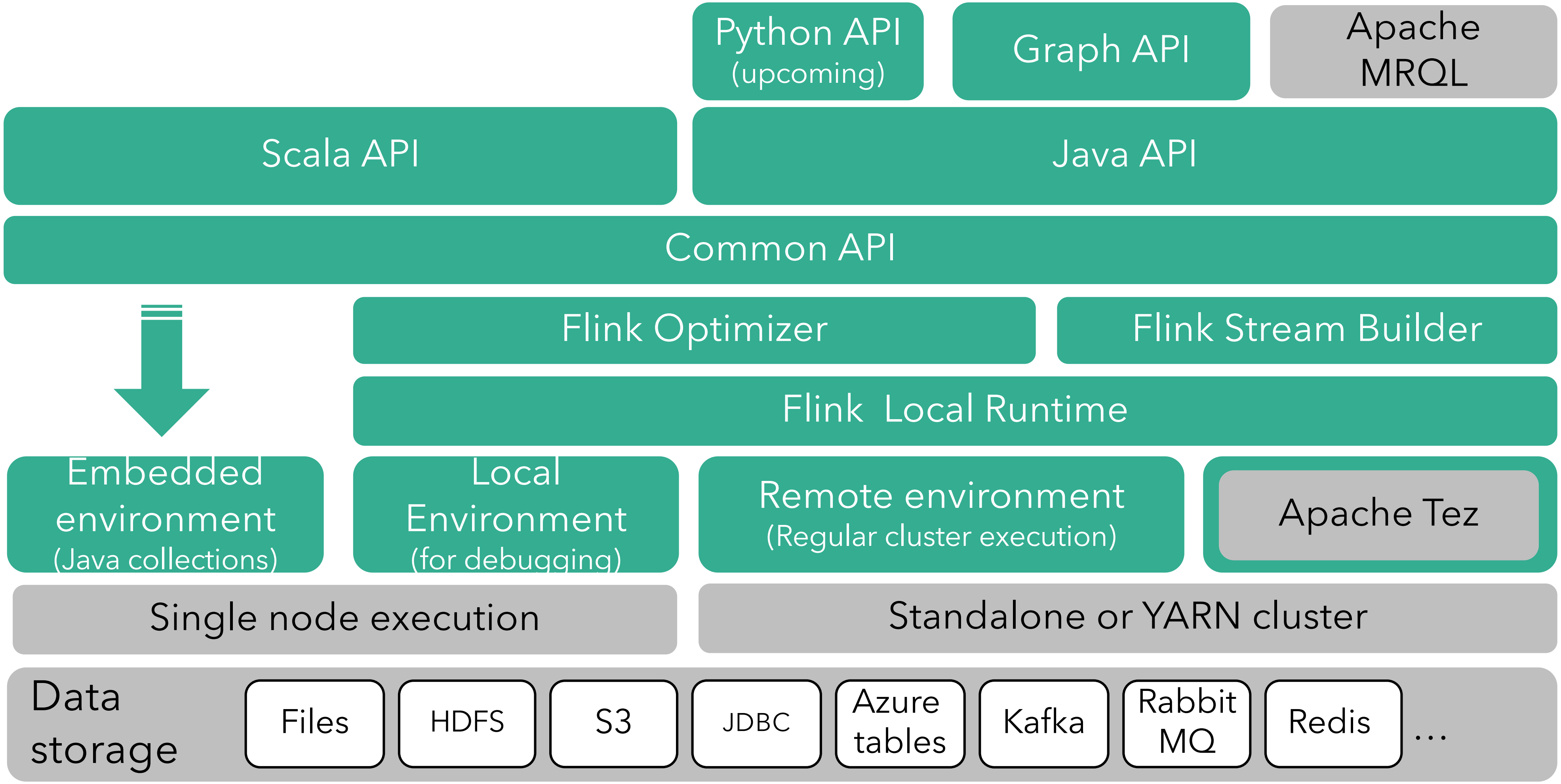


`StreamExecutionEnvironment#setBufferTimeout()`

*100 nodes x 8 slots

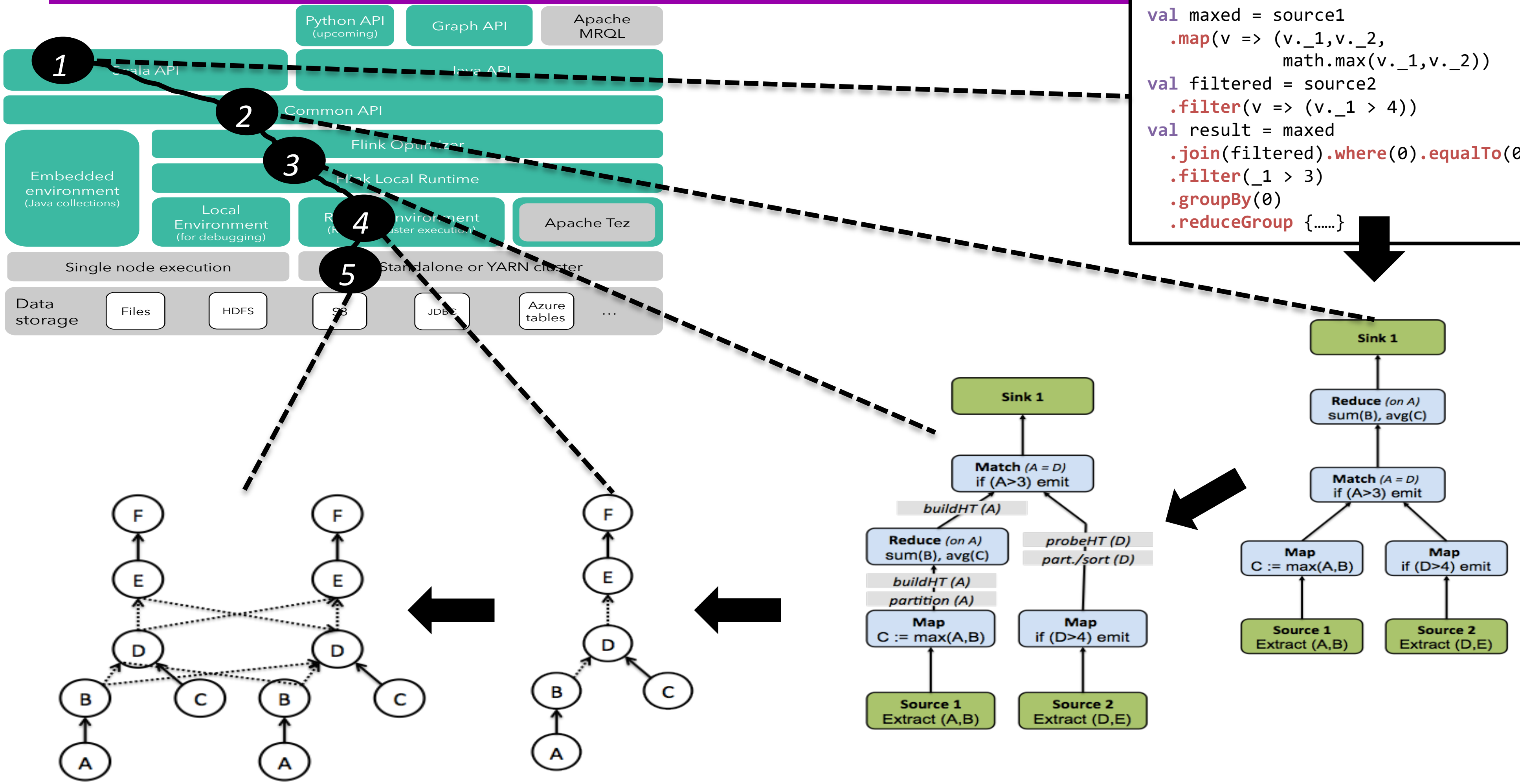
Program optimization

Recap: The Flink stack

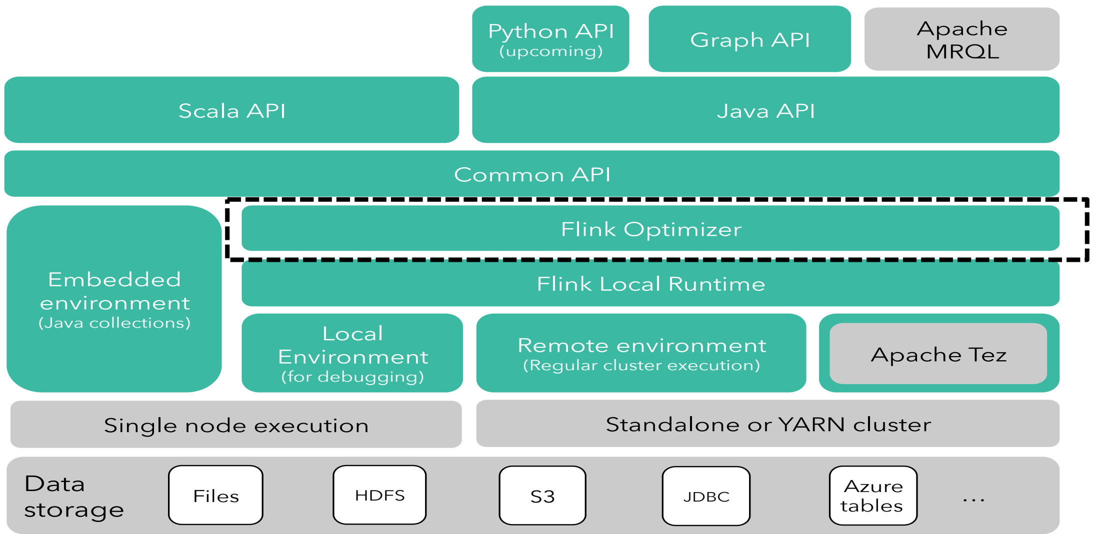


Program lifecycle

```
val source1 = …
val source2 = …
val maxed = source1
  .map(v => (v._1,v._2,
             math.max(v._1,v._2))
val filtered = source2
  .filter(v => (v._1 > 4))
val result = maxed
  .join(filtered).where(0).equalTo(0)
  .filter(_1 > 3)
  .groupBy(0)
  .reduceGroup {……}
```

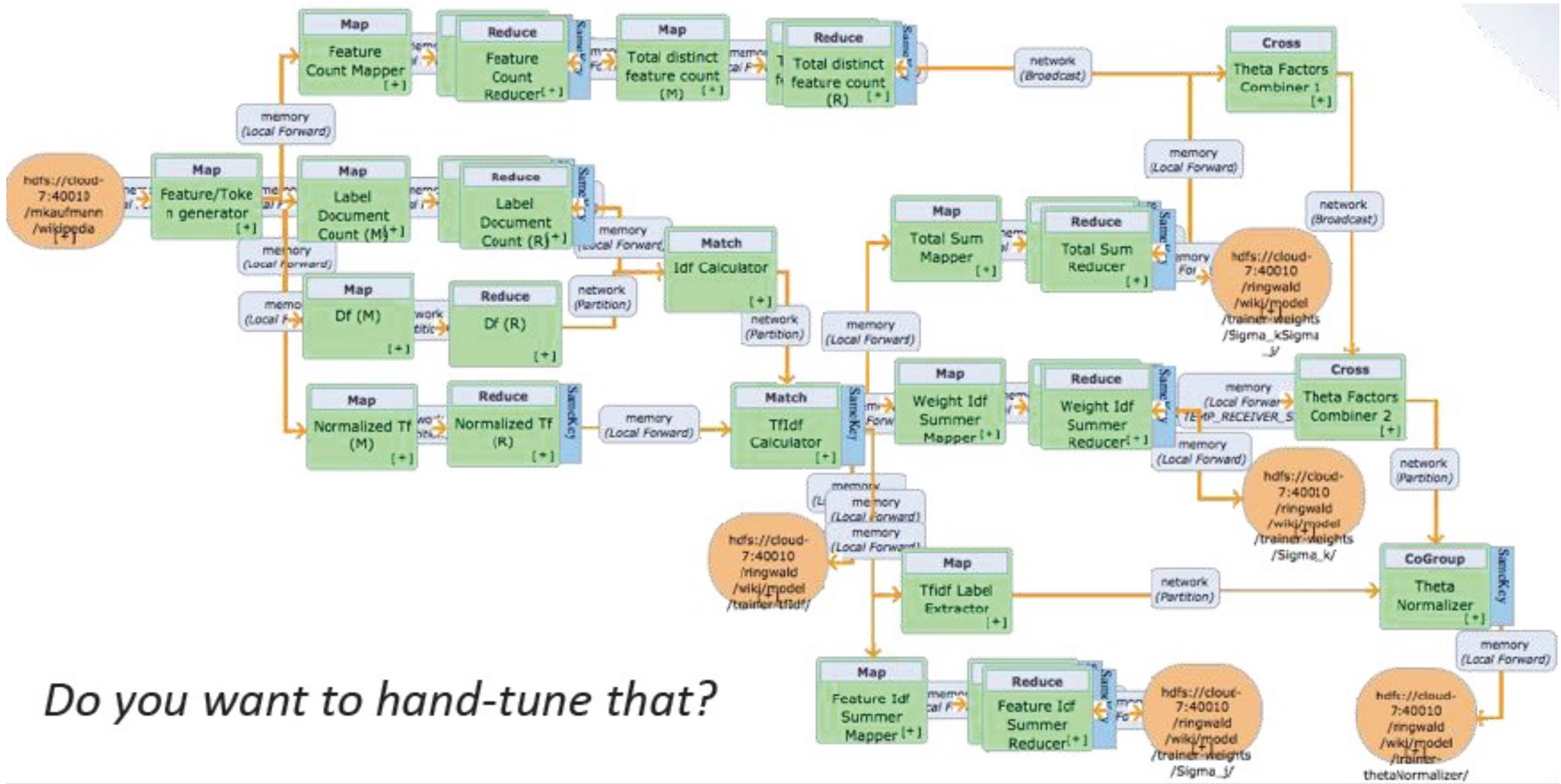


Flink Optimizer



- The optimizer is the component that selects an execution plan for a Common API program
- Think of an AI system manipulating your program for you ☺
- But don't be scared – it works
  - Relational databases have been doing this for decades – Flink ports the technology to API-based systems

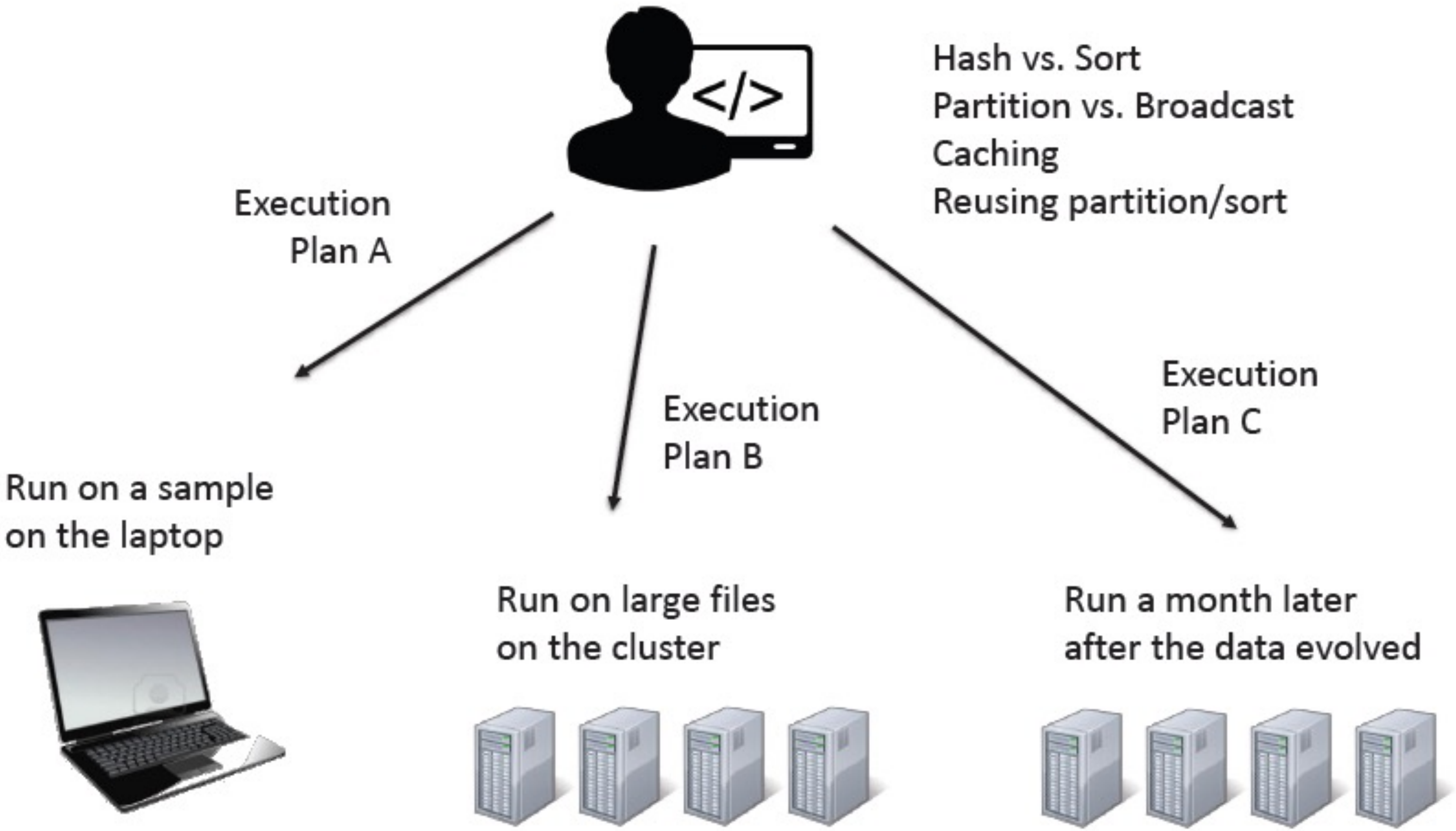
Optimization/auto-tuning – A Key design feature of Flink from its VERY BEGINNING



Do you want to hand-tune that?

F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, J.C. Freytag:
Peeking into the optimization of data flow programs with MapReduce-style UDFs. ICDE 2013: 1292-1295

Flink automatically optimizes Execution Plan of a program



Flink's Optimizer

- Inspired by optimizers of parallel database systems
  - Cost models and reasoning about interesting properties.
- Physical optimization follows cost-based approach
  - Select data shipping strategy (forward, partition, broadcast)
  - Local execution (sort merge join/ hash join)
  - Keep track of interesting properties such as sorting, grouping and partitioning
- Optimization of Flink programs more difficult than in the relational case:
  - No fully specified operator semantics due to UDFs
  - Unknown UDFs complicate estimating intermediate result sizes
  - No pre-defined schema present

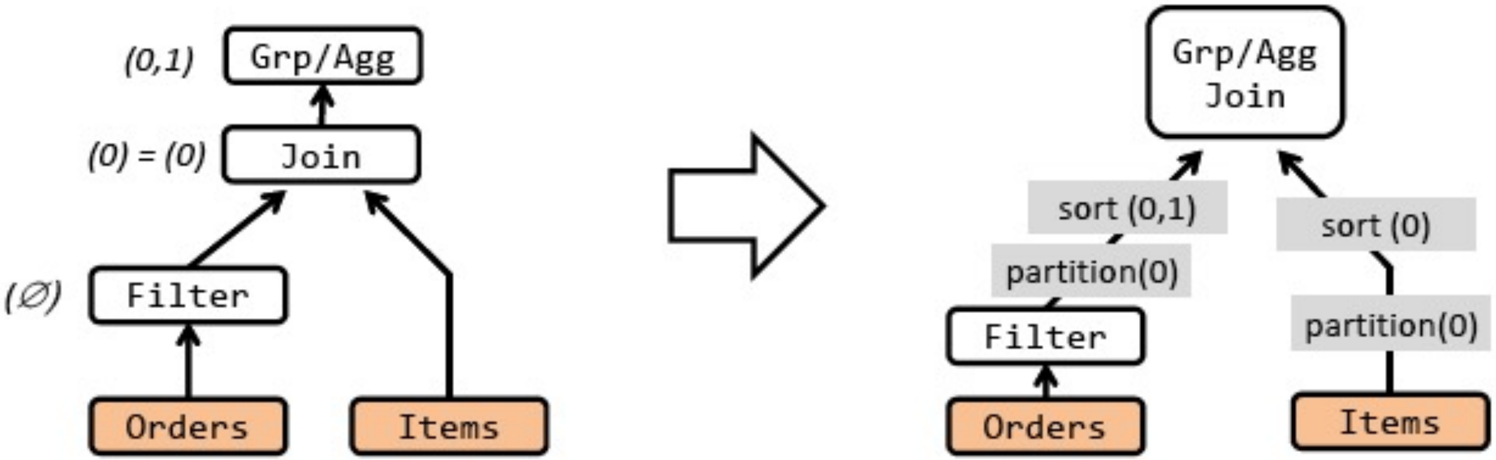
Example of optimizing a Flink program

```
val orders = …
val lineitems = …

val filteredOrders = orders
  .filter(o => dataFormat.parse(l.shipDate).after(date))
  .filter(o => o.shipPrio > 2)

val lineitemsOfOrders = filteredOrders
  .join(lineitems)
  .where(“orderId”).equalTo(“orderId”)
  .apply((o,l) => new SelectedItem(o.orderDate, l.extdPrice))

val priceSums = lineitemsOfOrders
  .groupBy(“orderDate”).sum(“l.extdPrice”);
```

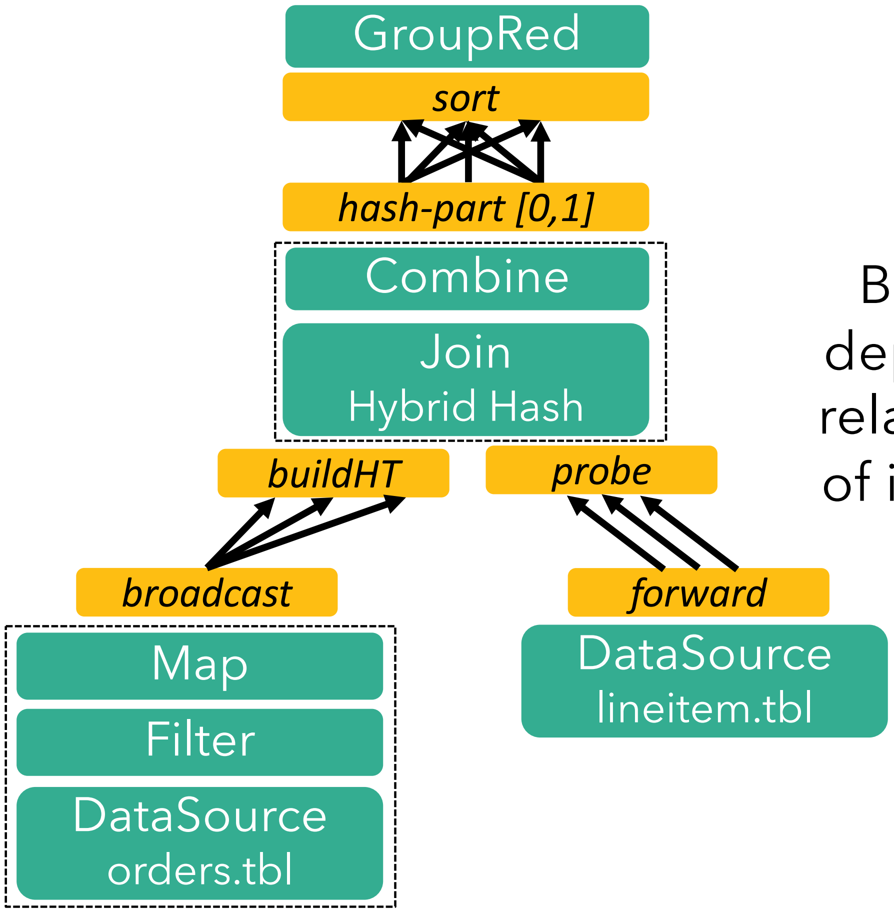
Another Optimization Example

```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: double, )
case class PricedOrder(id, priority, price)
```

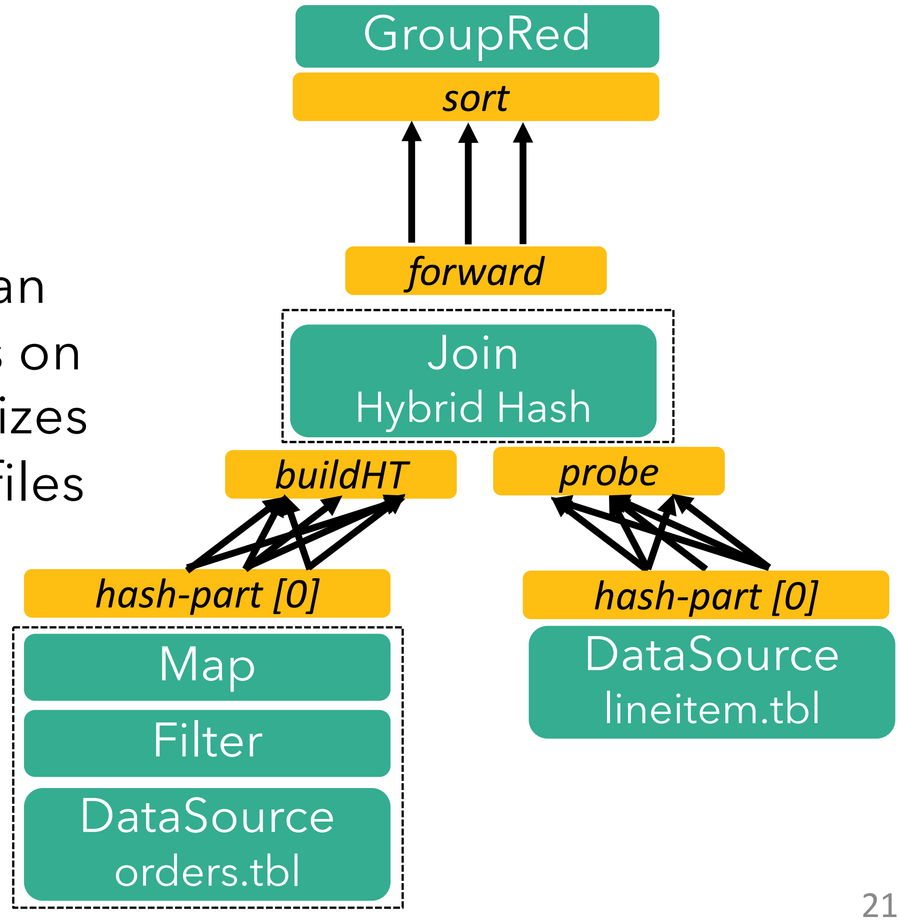
```
val orders = DataSource(...)
val items = DataSource(...)

val filtered = orders filter { ... }

val prio = filtered join items where { _.id } isEqualTo { _.id }
                        map {(o,li) => PricedOrder(o.id, o.priority, li.price)}

val sales = prio groupBy {p => (p.id, p.priority)} aggregate ({_.price},SUM)
```



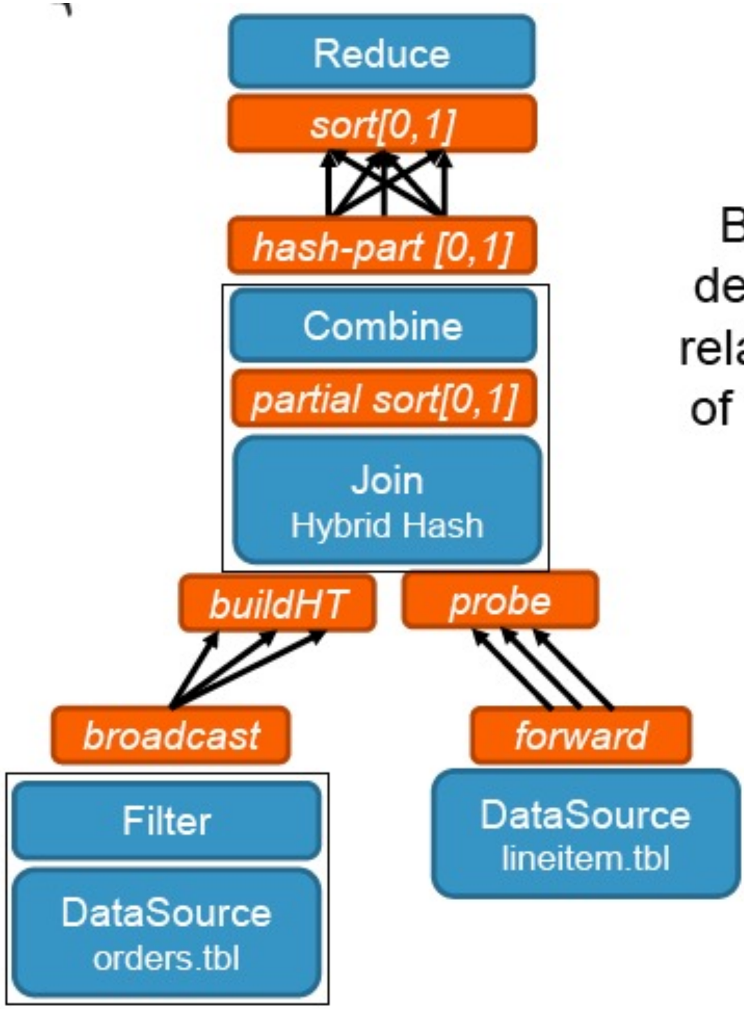
Two execution plans



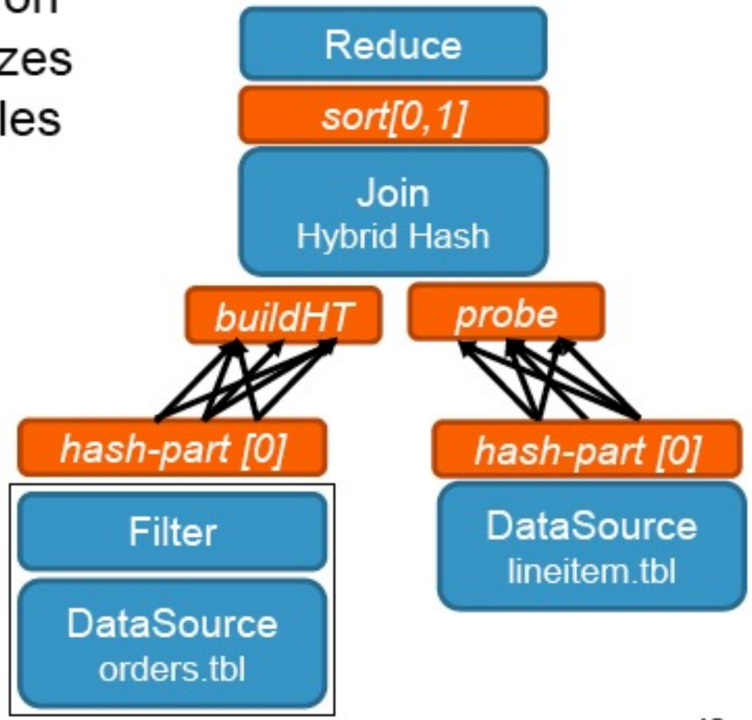
Best plan depends on relative sizes of input files



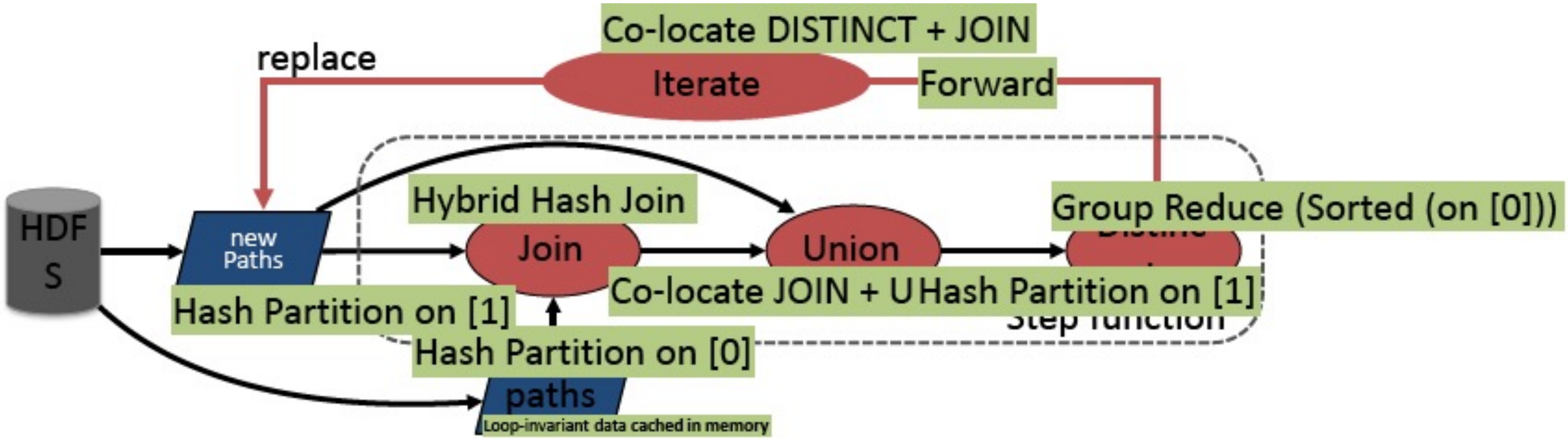
Data Flow Optimizer



Best plan depends on relative sizes of input files



Example: Flink's Optimization on Transitive Closure



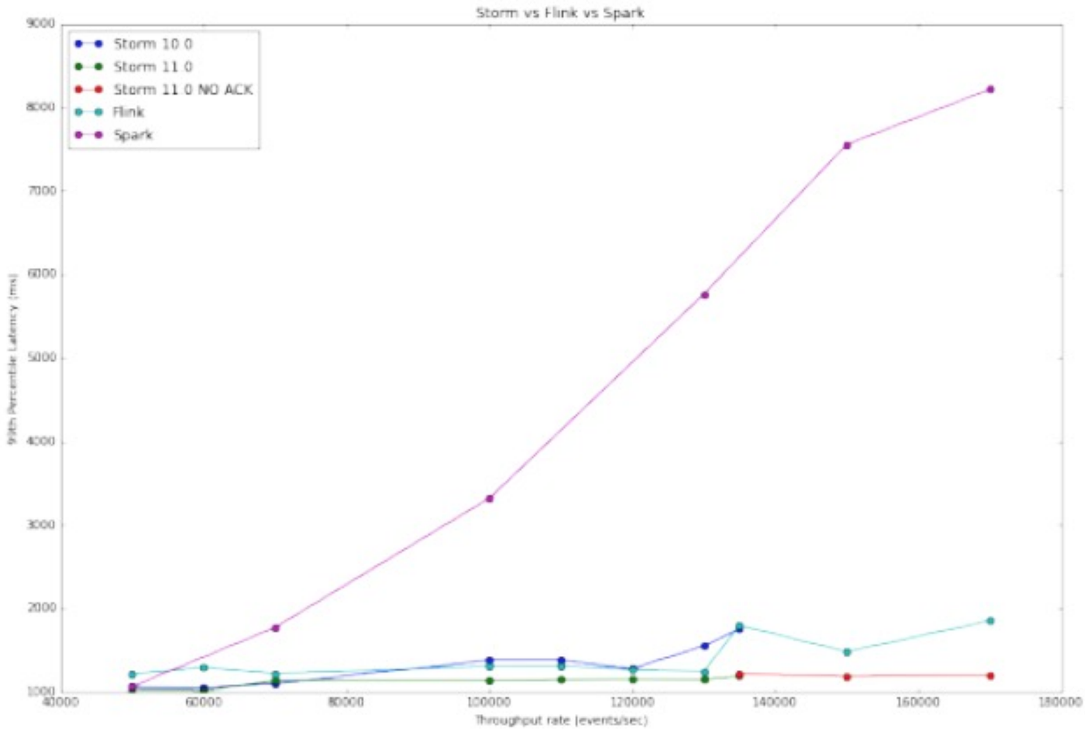
- What you write is ***not*** what is executed
- No need to hardcode execution strategies

- Flink Optimizer decides:
  - Pipelines and dam/barrier placement
  - Sort- vs. hash- based execution
  - Data exchange (partition vs. broadcast)
  - Data partitioning steps
  - In-memory caching

More Examples of Optimization

- Task chaining
  - Coalesce map/filter/etc tasks
- Join optimizations
  - Broadcast/partition, build/probe side, hash or sort-merge
- Interesting properties
  - Re-use partitioning and sorting for later operations
- Automatic caching
  - E.g., for iterations

Yahoo! Benchmark Results (circa Dec 2015)

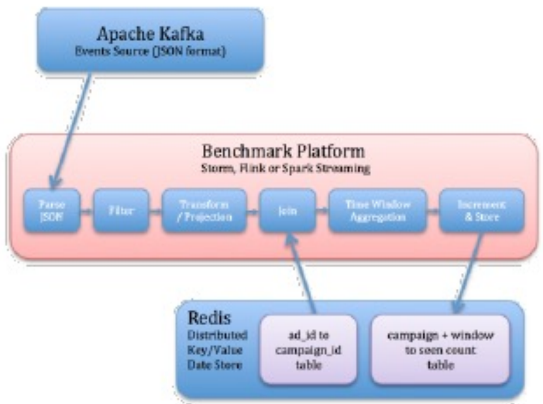


Source: http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Performed by Yahoo! Engineering, Dec 16, 2015

[..]Storm 0.10.0, 0.11.0-SNAPSHOT and Flink 0.10.1 show sub- second latencies at relatively high throughputs[..]. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.

Flink achieves highest throughput with competitive low latency!



dataArtisan's Benchmark Results

Streaming

	2-node	4-node	8-node
Storm	408K	696K	992K
Spark	379K	642K	912K
Flink	1230K	1260K	1260K

	2-node	4-node	8-node
Spark	365K	632K	947K
Flink	851K	1128K	1190K

Windowed Aggregations / Joins

Flink consistently outperforms other streaming engines in throughput and latency

Batch

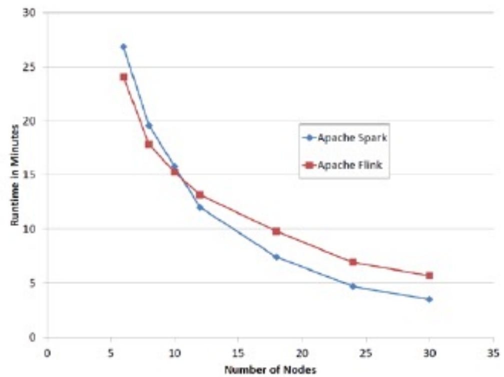


Figure 10: k-means Strong Scaling experiments for Spark and Flink in 200 GB of generated data with 100 dimensions and k=*10* clusters

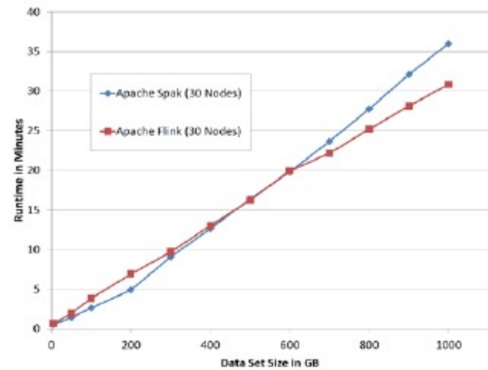
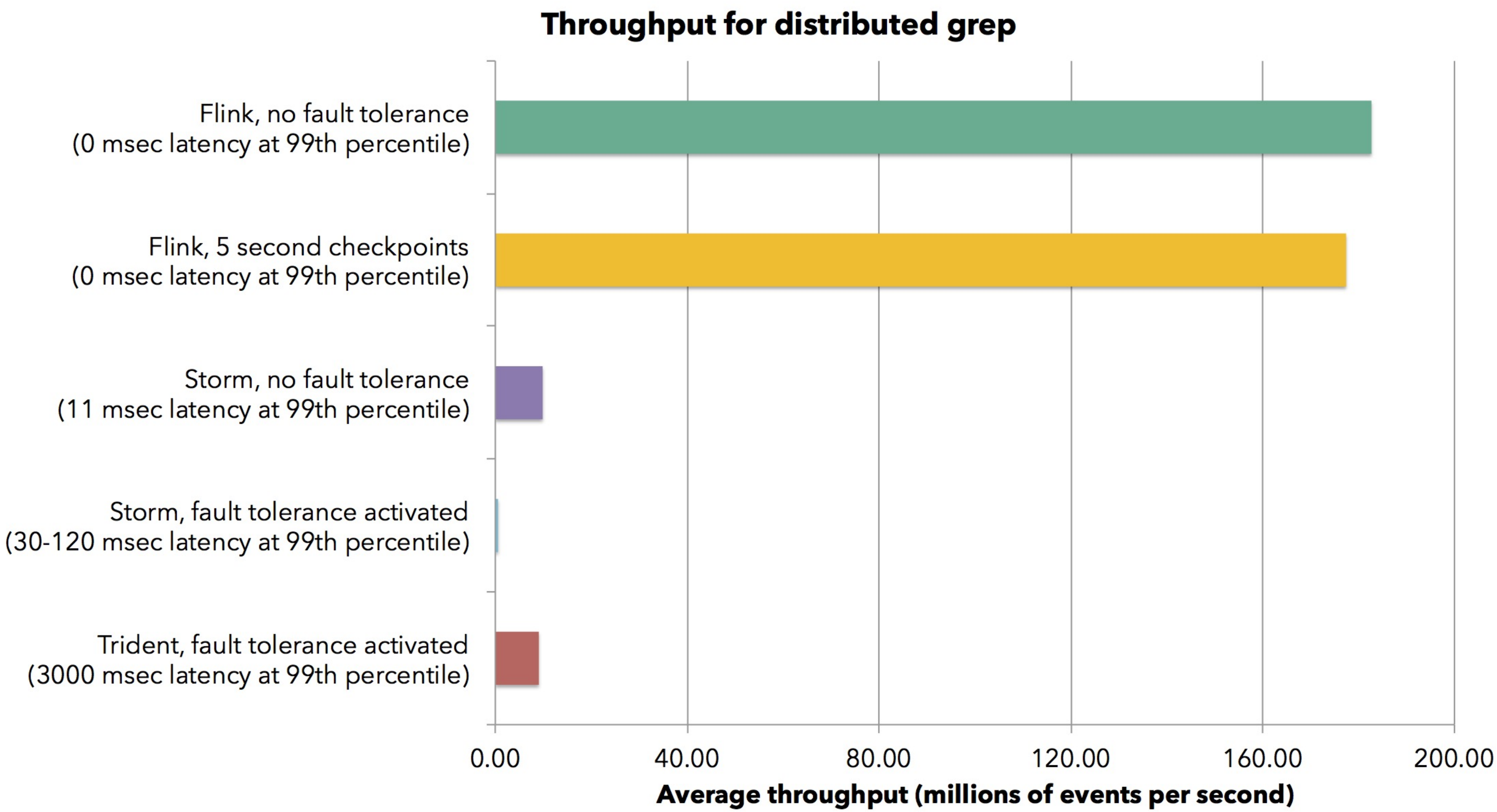


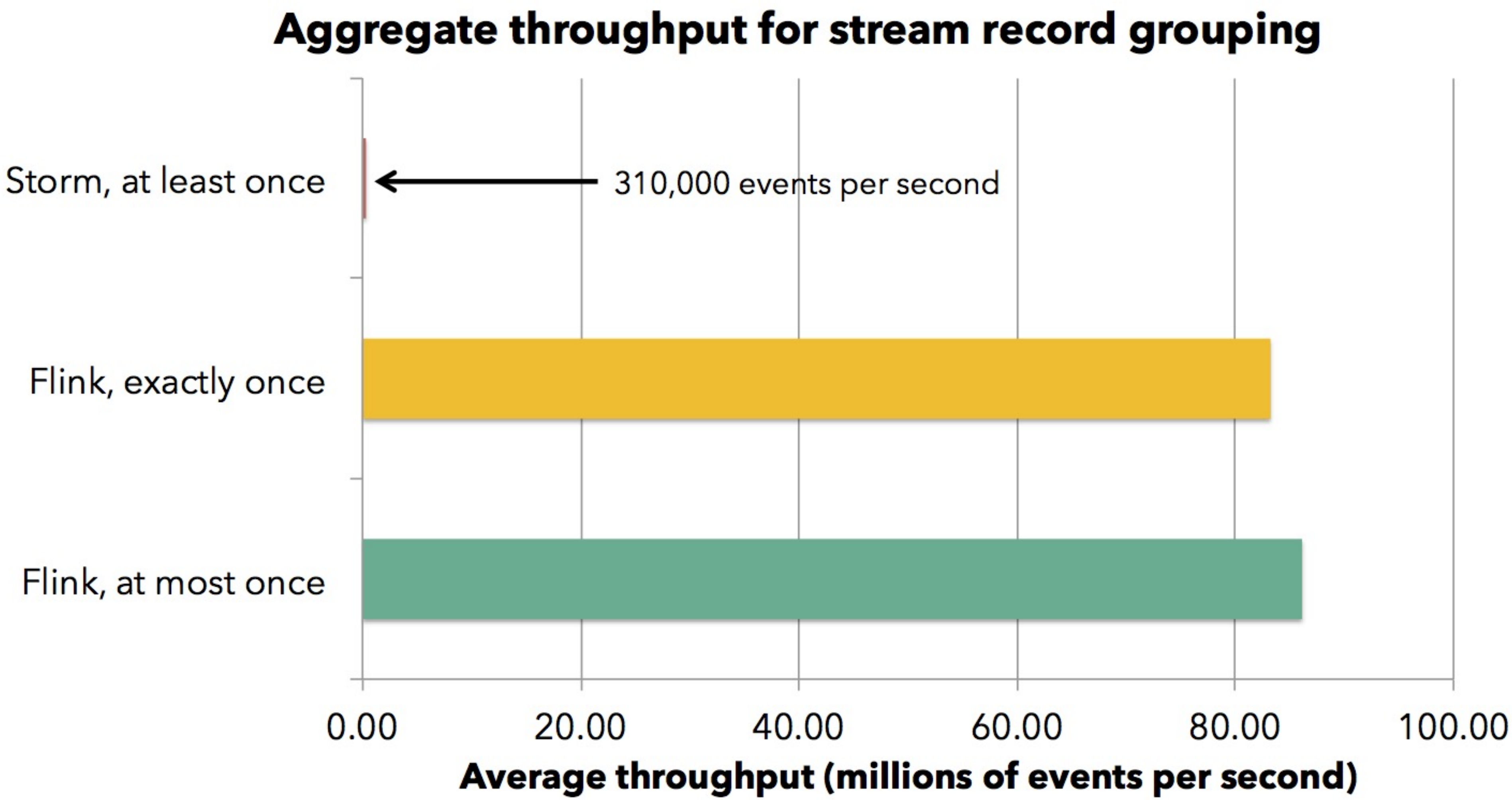
Figure 11: k-means Production Scaling experiments for Spark and Flink on 30 nodes with k=*30*

Iterative Algorithms

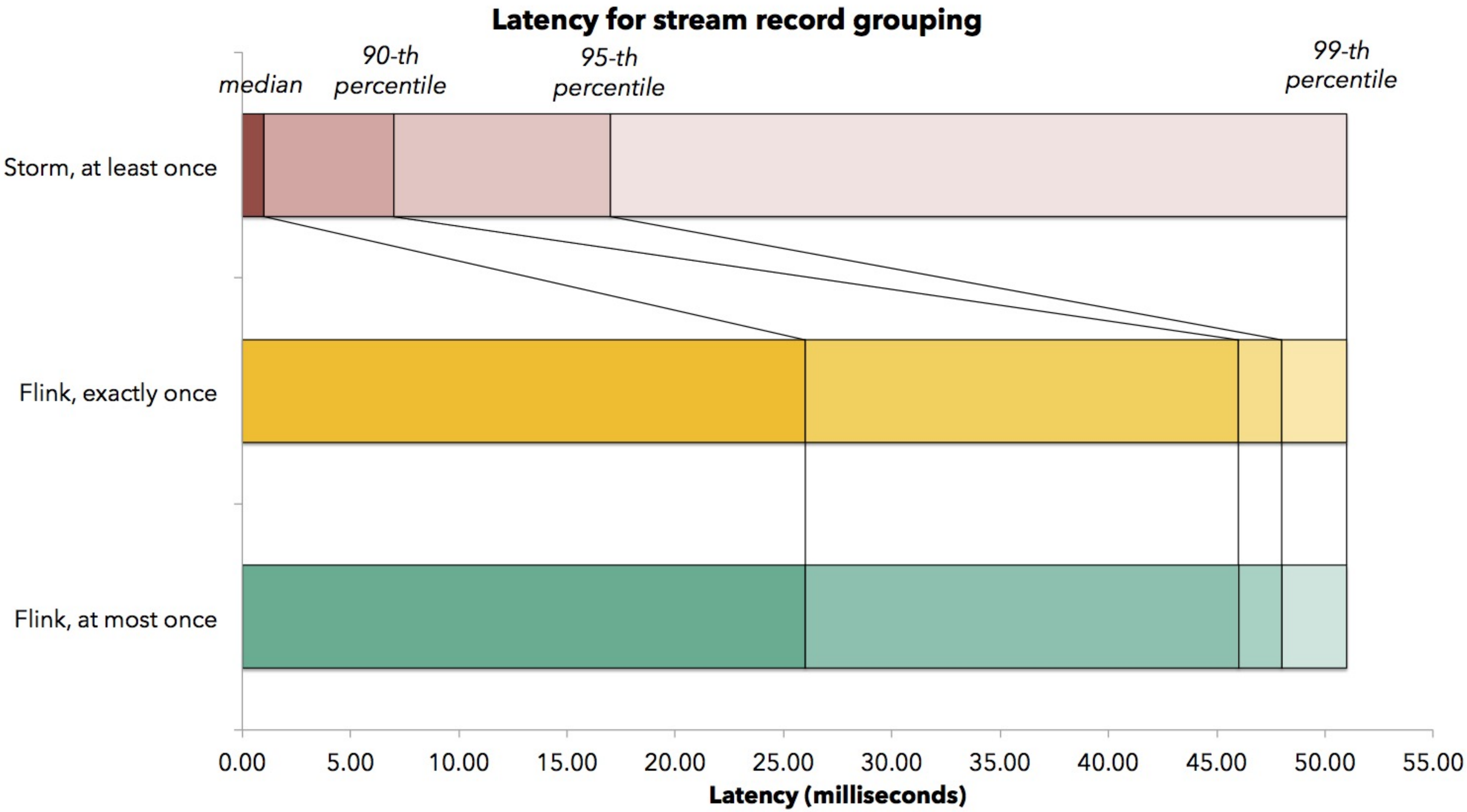
Show me the (Performance) Numbers !



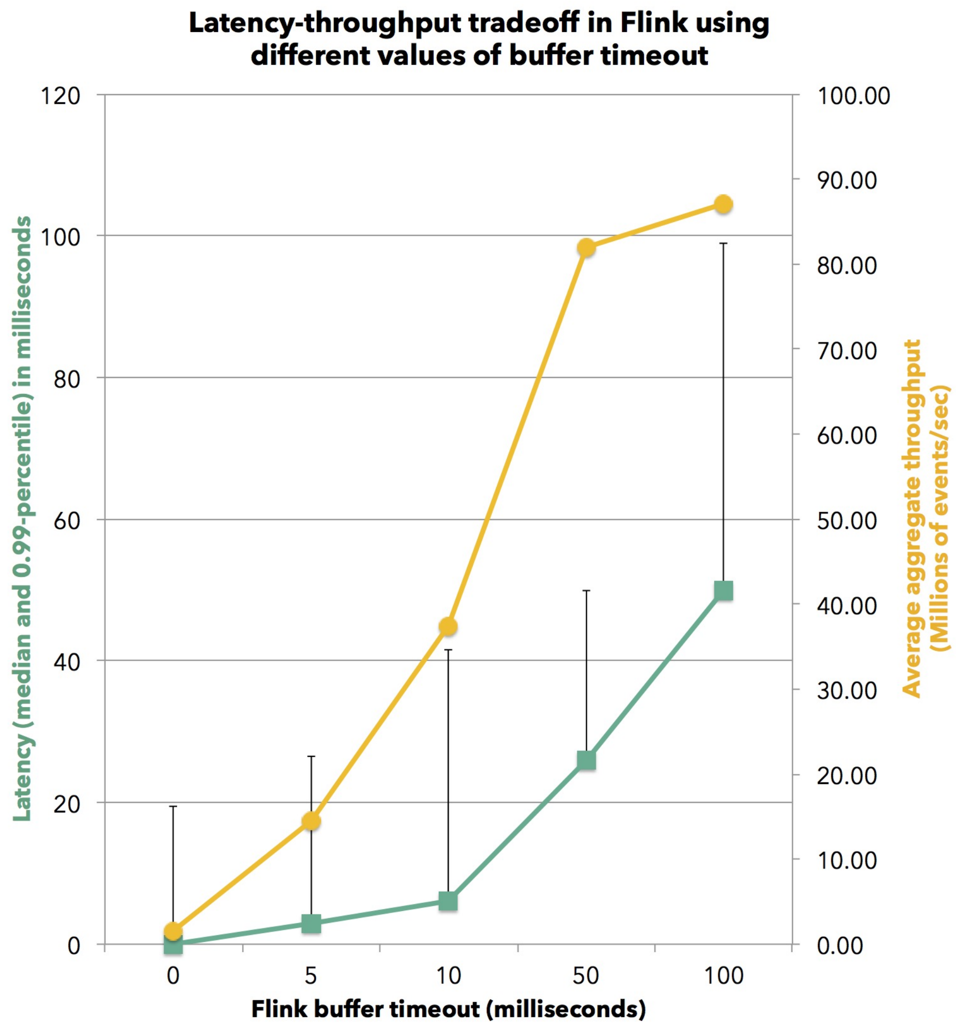
Show me the (Performance) Numbers !



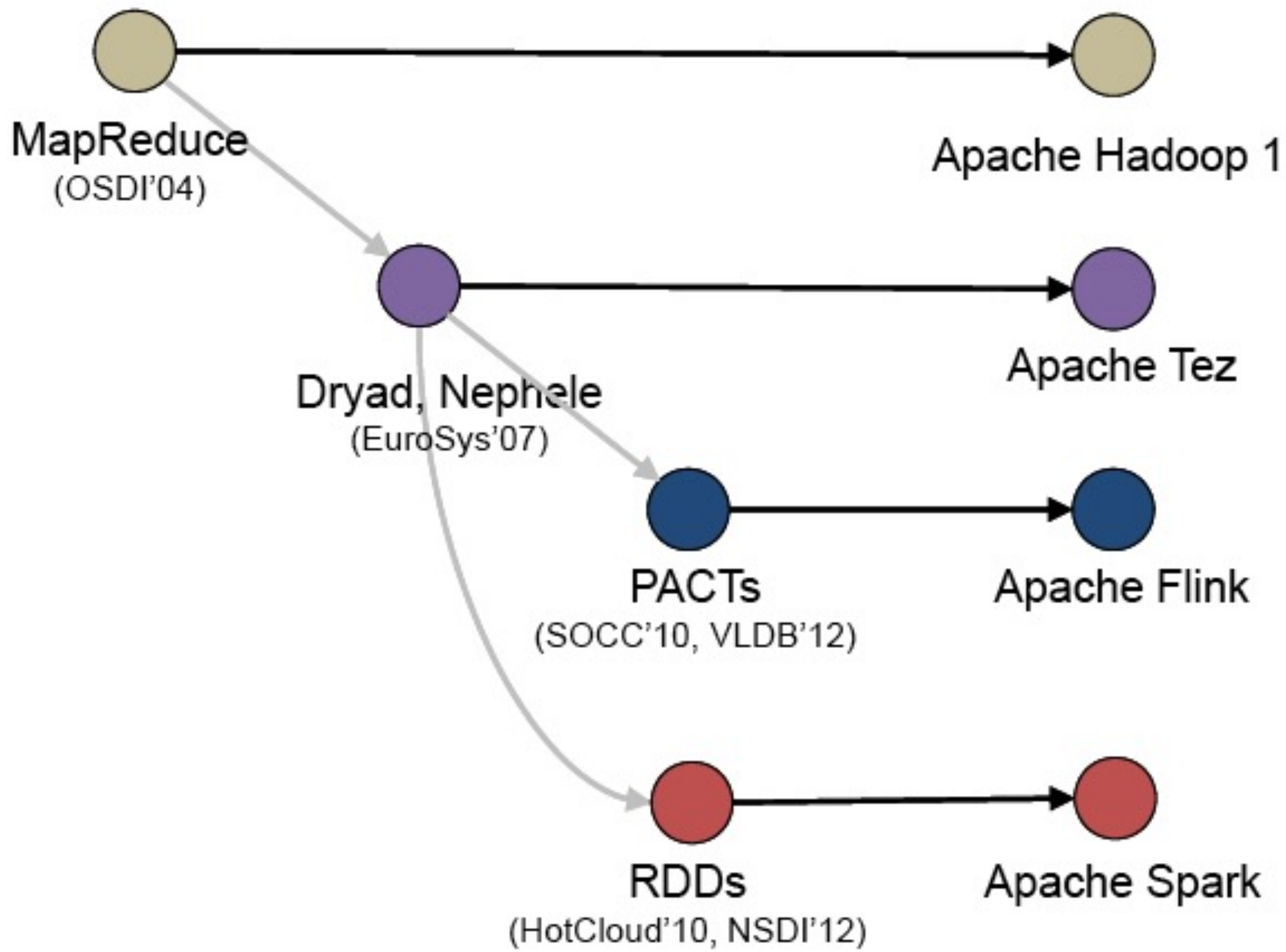
Show me the (Performance) Numbers !



Show me the (Performance) Numbers !



Comparing Engine Paradigms & Systems



Engine Comparison

	Hadoop MapReduce	TEZ	Spark	Flink
API	MapReduce on k/v pairs	k/v pair Readers/Writers	Transformations on k/v pair collections	Iterative transformations on collections
Paradigm	MapReduce	DAG	RDD	Cyclic dataflows
Optimization	none	none	Optimization of SQL queries	Optimization in all APIs
Execution	Batch sorting	Batch sorting and partitioning	Batch with memory pinning	Stream with out-of-core algorithms

Batch Comparison

API	low-level	high-level	high-level
Data Transfer	batch	batch	pipelined & batch
Memory Management	disk-based	JVM-managed	Active managed
Iterations	file system cached	in-memory cached	streamed
Fault tolerance	task level	task level	job level
Good at	massive scale out	data exploration	heavy backend & iterative jobs
Libraries	many external	built-in & external	evolving built-in & external

Streaming Comparison

Streaming	"true"	mini batches	"true"
API	low-level	high-level	high-level
Fault tolerance	tuple-level ACKs	RDD-based (lineage)	coarse checkpointing
State	not built-in	external	internal
Exactly once	at least once	exactly once	exactly once
Windowing	not built-in	restricted	flexible
Latency	low	medium	low
Throughput	medium	high	high

Deployment and Process Model

Diverse Deployment Scenarios

- Many different deployment scenarios
  - Yarn
  - Mesos
  - Docker/Kubernetes
  - Standalone
  - Etc.

Flink Improvement Proposal 6

- Introduce generic building blocks
- Compose blocks for different scenarios
- Effort started by:

Alibaba Group dataArtisans

Flip-6 design document:
https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65147077

Flink's Revamped Distributed Architecture

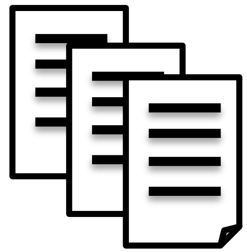
- Motivation
  - Resource Elasticity
  - Support for Different Deployments
  - REST interface for Client-Cluster communications
- Introduce generic Building Blocks
- Compose blocks for different scenarios

Different Usage Patterns

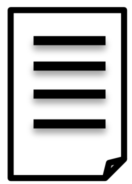
- Few long running vs. many short running jobs
  - Overhead of starting a Flink cluster
- Job isolation vs. sharing resources
  - Allowing to define per job credentials & secrets
  - Efficient resource utilization by sharing them

Job & Session Mode

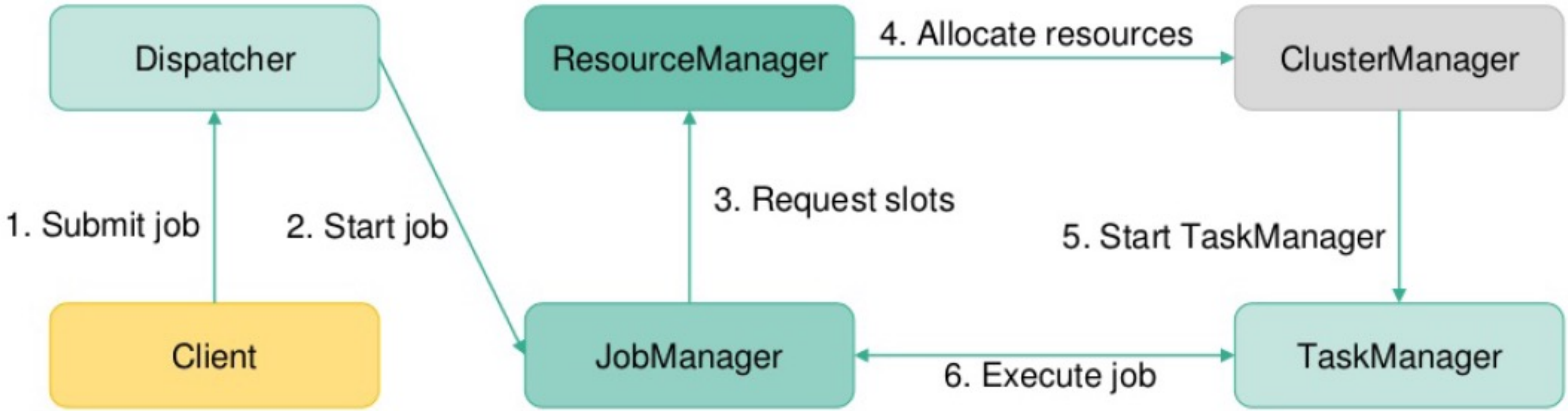
- Session mode
  - Shared cluster for multiple jobs
  - Resources can be shared across jobs
  - Cluster deployment and job submission separate actions



- Job mode
  - Dedicated cluster for a single job
  - Job should be part of the cluster deployment



Revamped distributed architecture



- Support for full resource elasticity
- Application parallelism can be dynamically changed

The Building Blocks

ResourceManager

- ClusterManager-specific
- May live across jobs
- Manages available Containers/TaskManagers
- Acquires / releases resources

Dispatcher

- Lives across jobs
- Touch-point for job submissions
- Spawns JobManagers

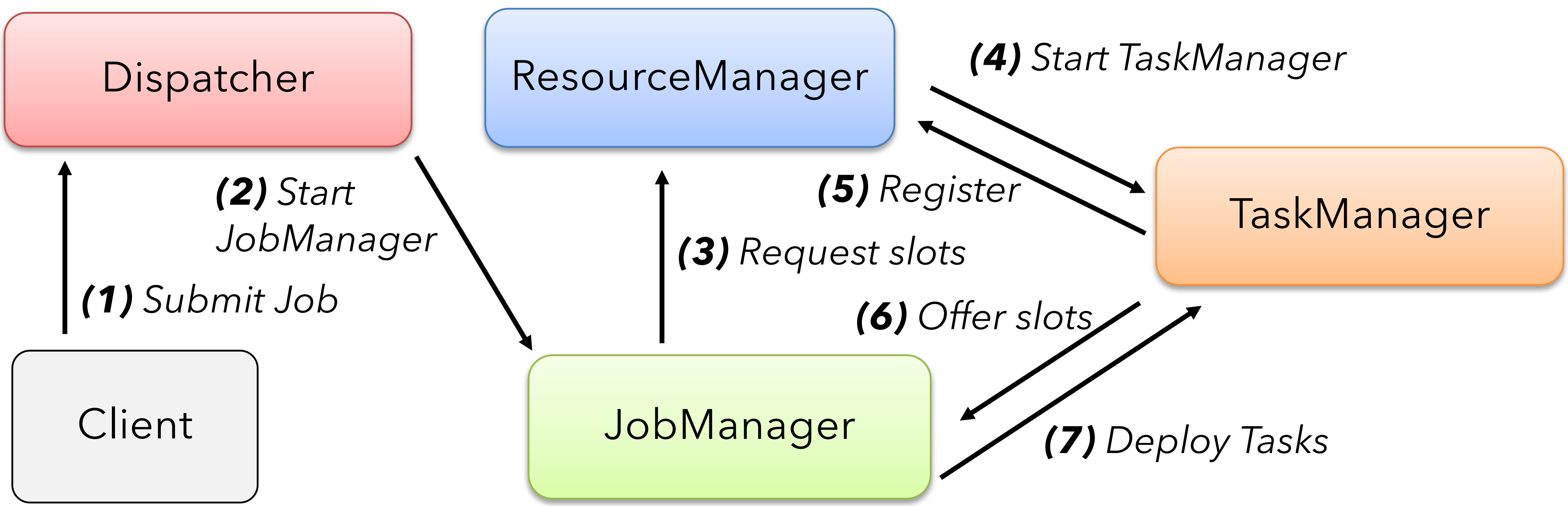
JobManager

- Single job only, started per job
- Thinks in terms of "task slots"
- Deploys and monitors job/task execution

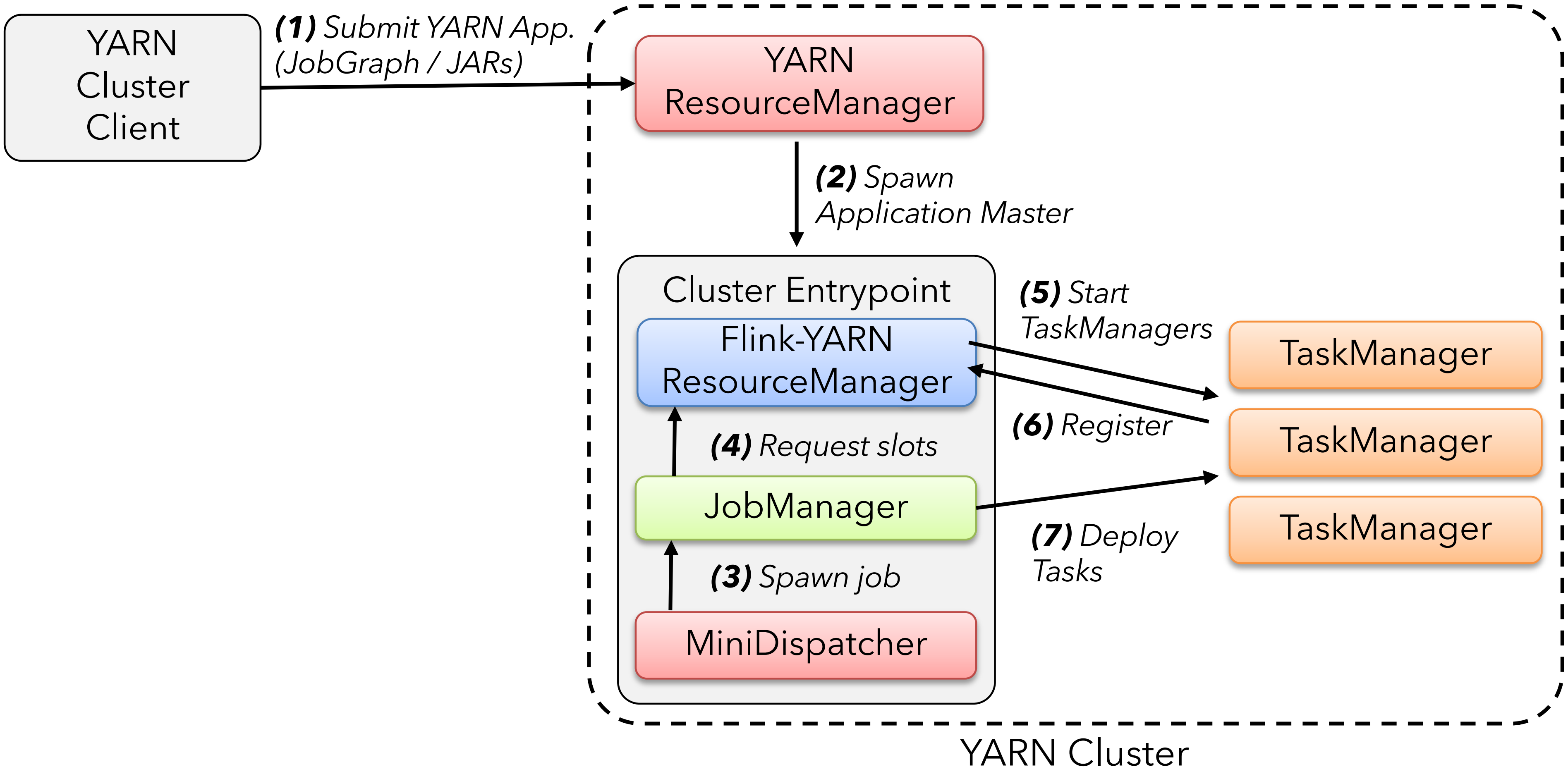
TaskManager

- Registers at ResourceManager
- Gets tasks from one or more JobManagers

The Building Blocks



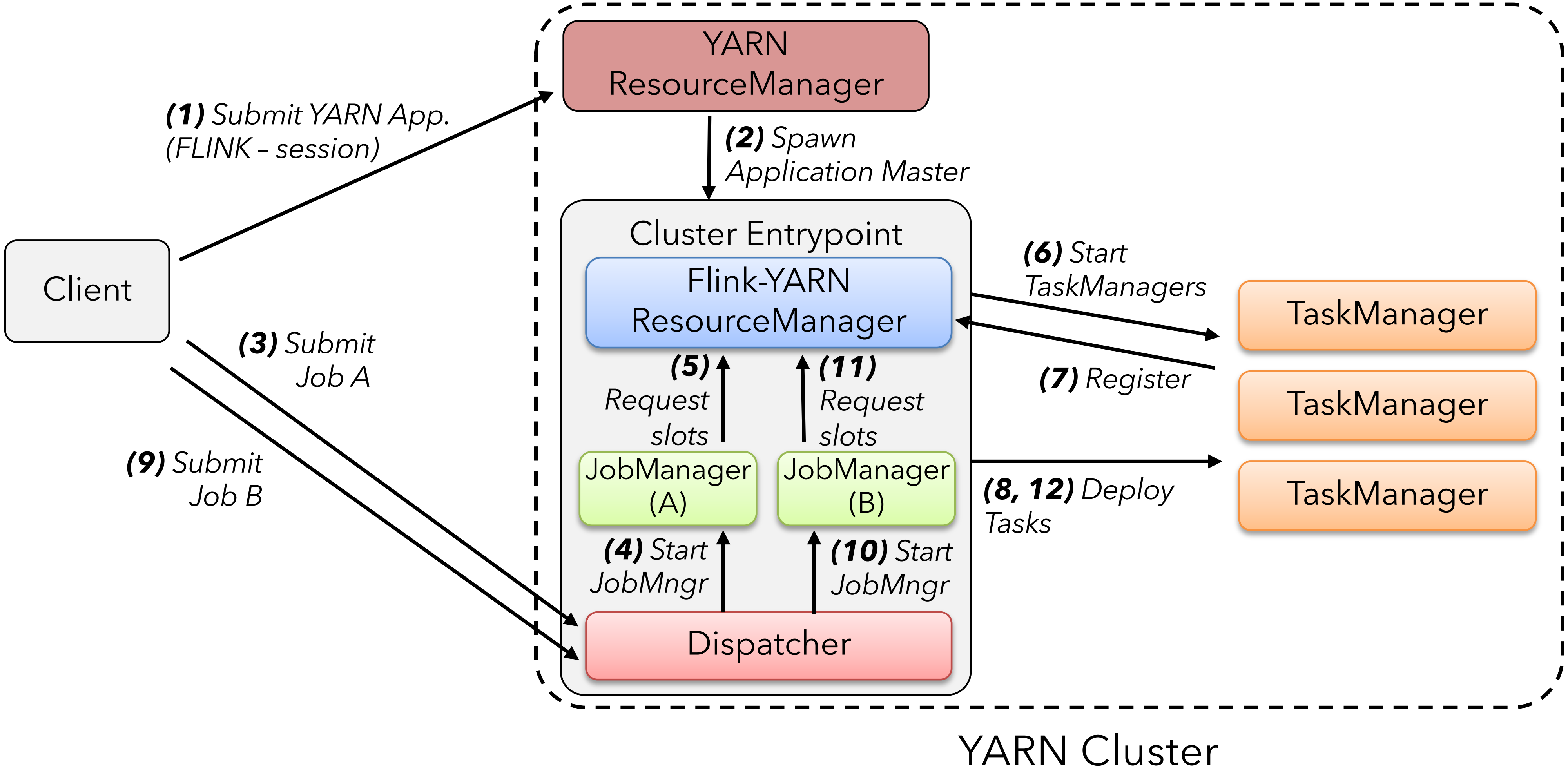
Building YARN PER-JOB MODE



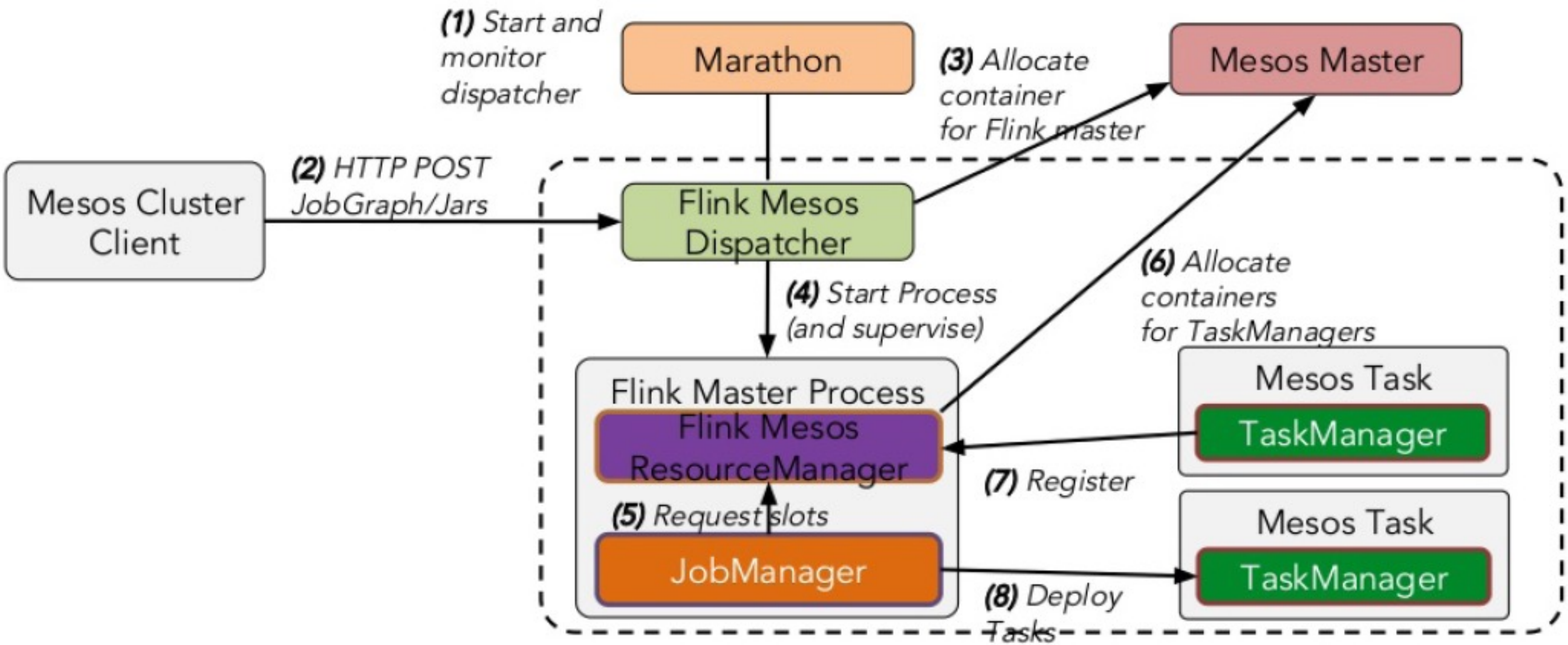
Differences to old YARN Per-job mode

- User JARs in classpath of all components
  - Fewer class loading issues
- Dynamic resources allocation
  - No longer necessary to specify number of containers at start-up
- No two phase job submission

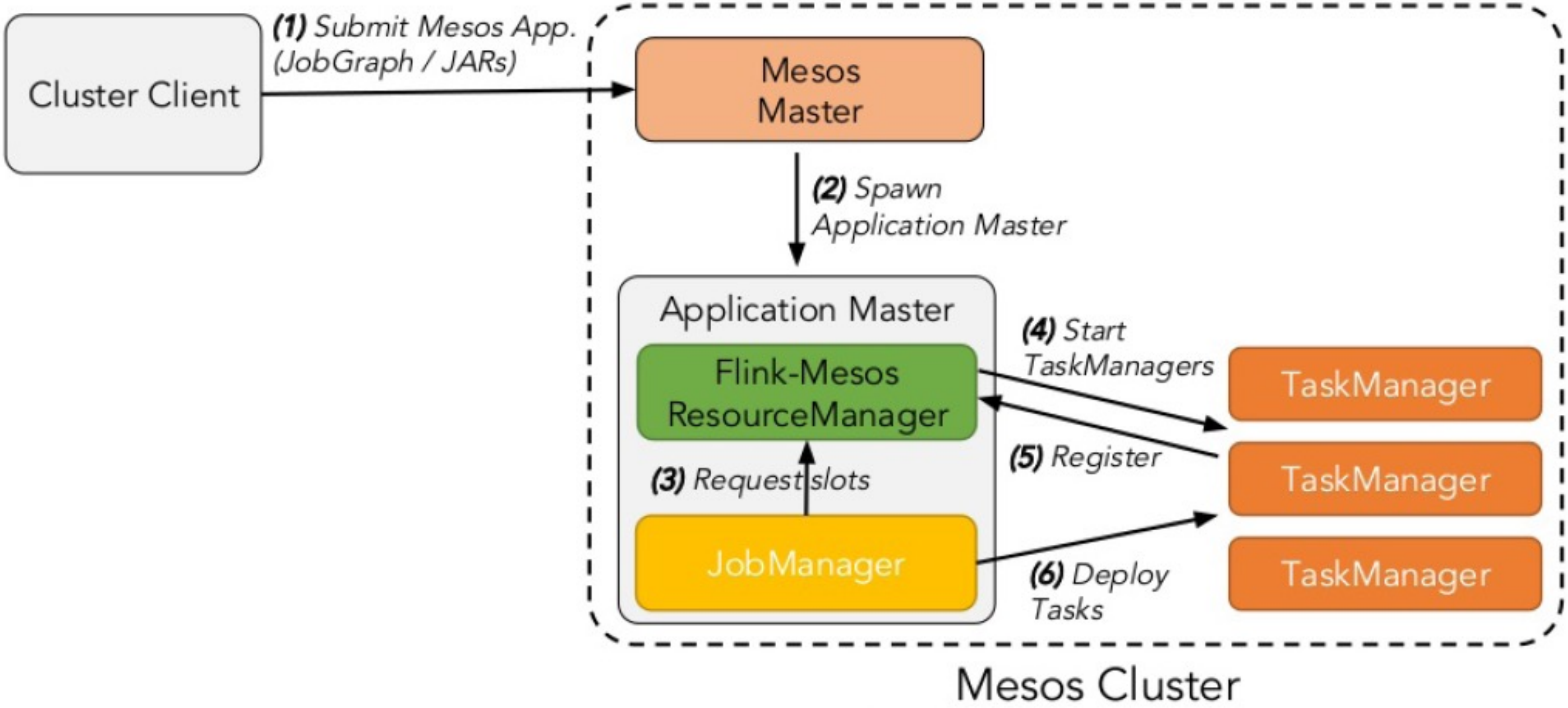
BUILDING YARN Session MODE



Flink Mesos Integration

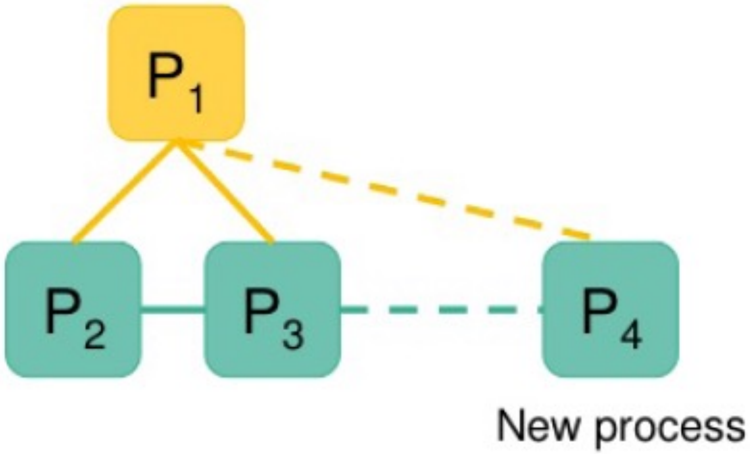


Building Flink-on-Mesos (Job mode)

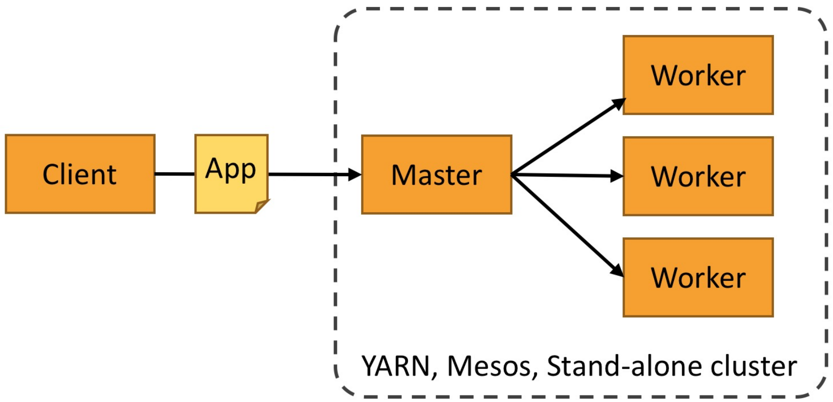


Flink as a library (and still as a framework)

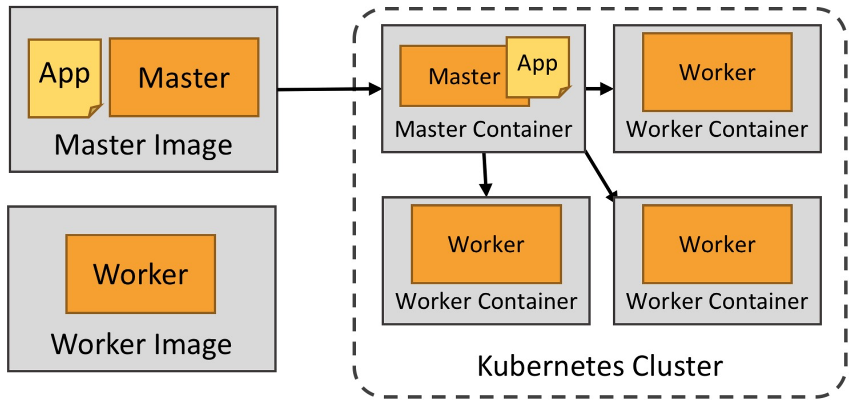
- Deploying Flink applications should be as easy as starting a process
- Bundle application code and Flink into a single image
- Process connects to other application processes and figures out its role
- Removing the cluster out of the equation



Deploying Flink as a Framework vs. as a Library



Framework Deployment



Library Deployment

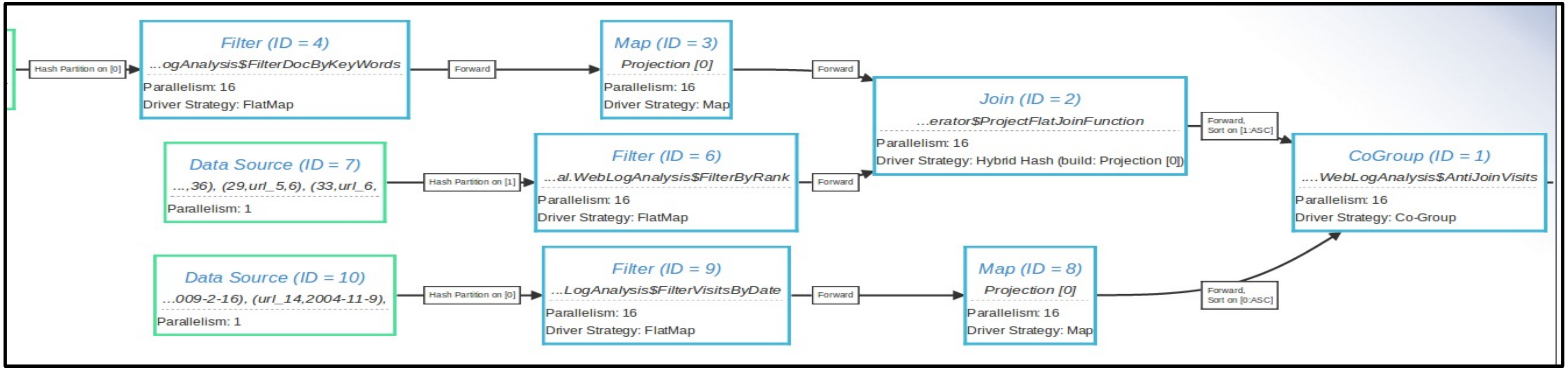
Standing Processes / Endpoints,
Dynamic Control over Resources

Long running application
under the control of your
container manager

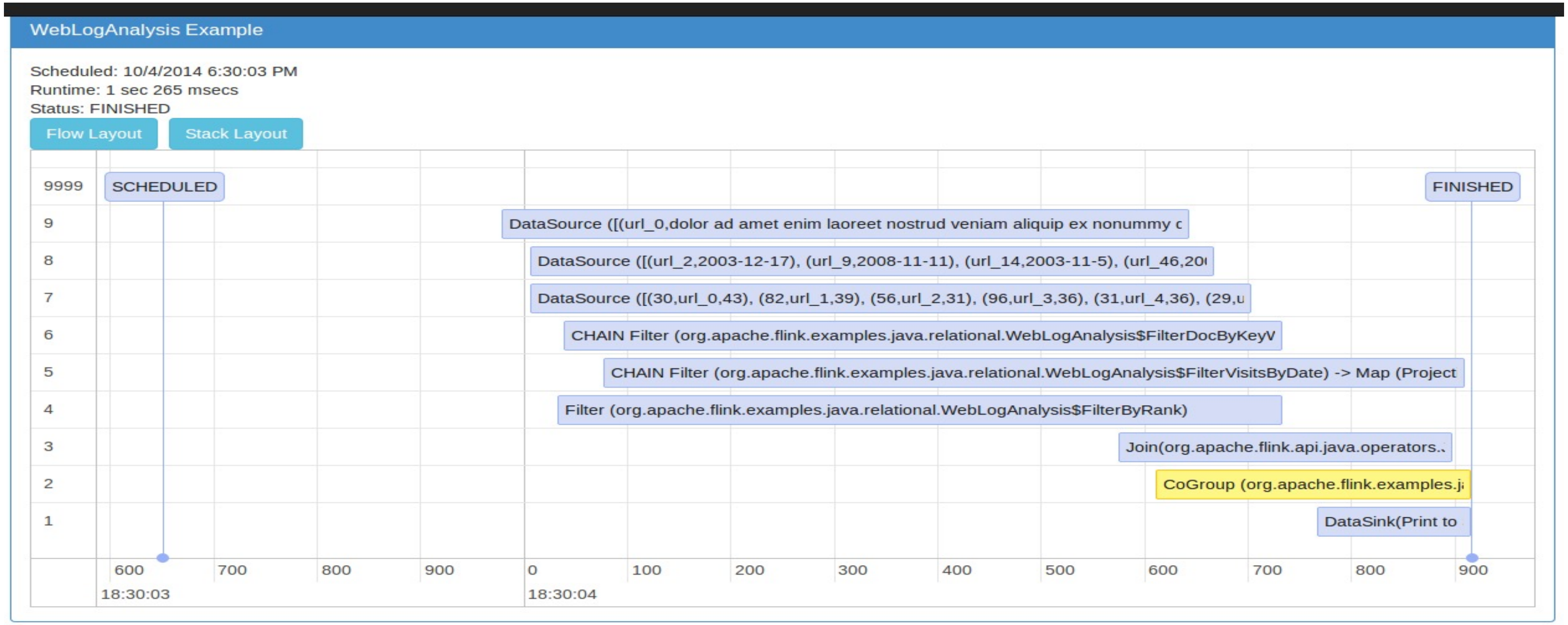
Deployment Model Wrap up

- New distributed architecture allows Flink to support many different deployment scenarios
- Flink now supports a native “job” mode as well as the “session” mode
- Support for full resource elasticity
- REST interface for easy cluster communication

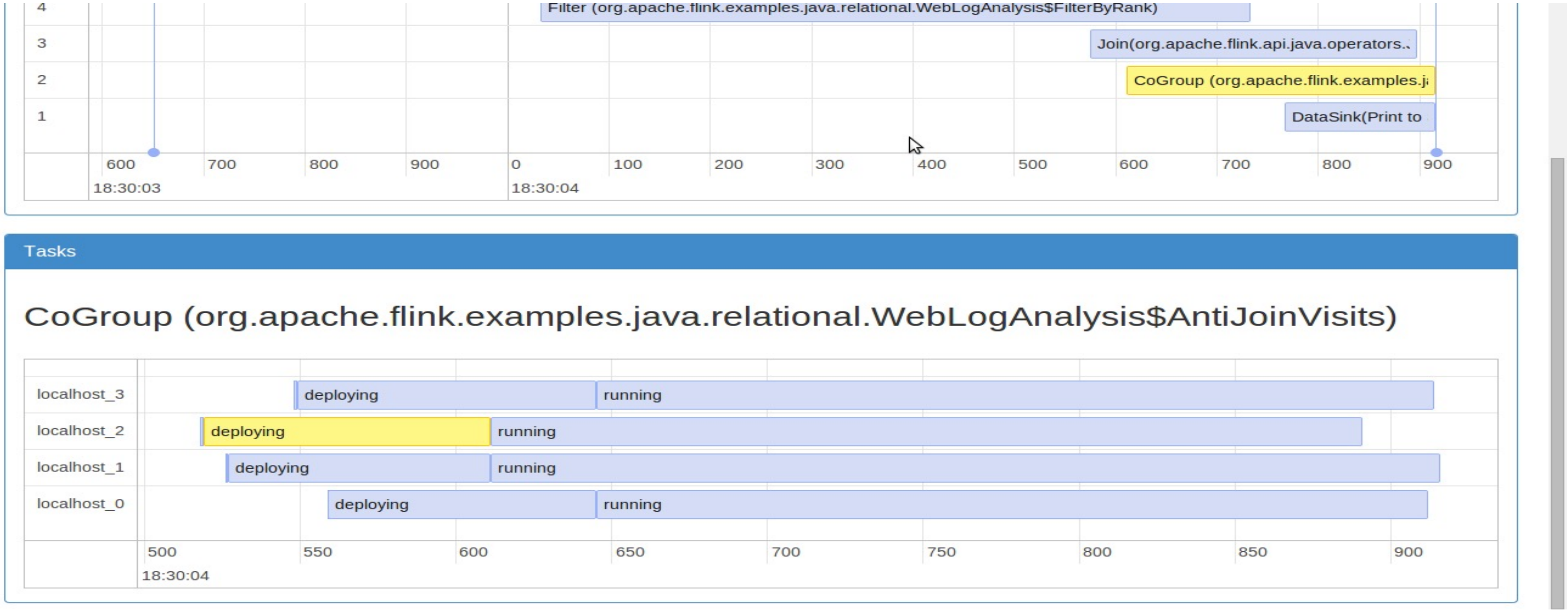
Visualization tools



Visualization tools

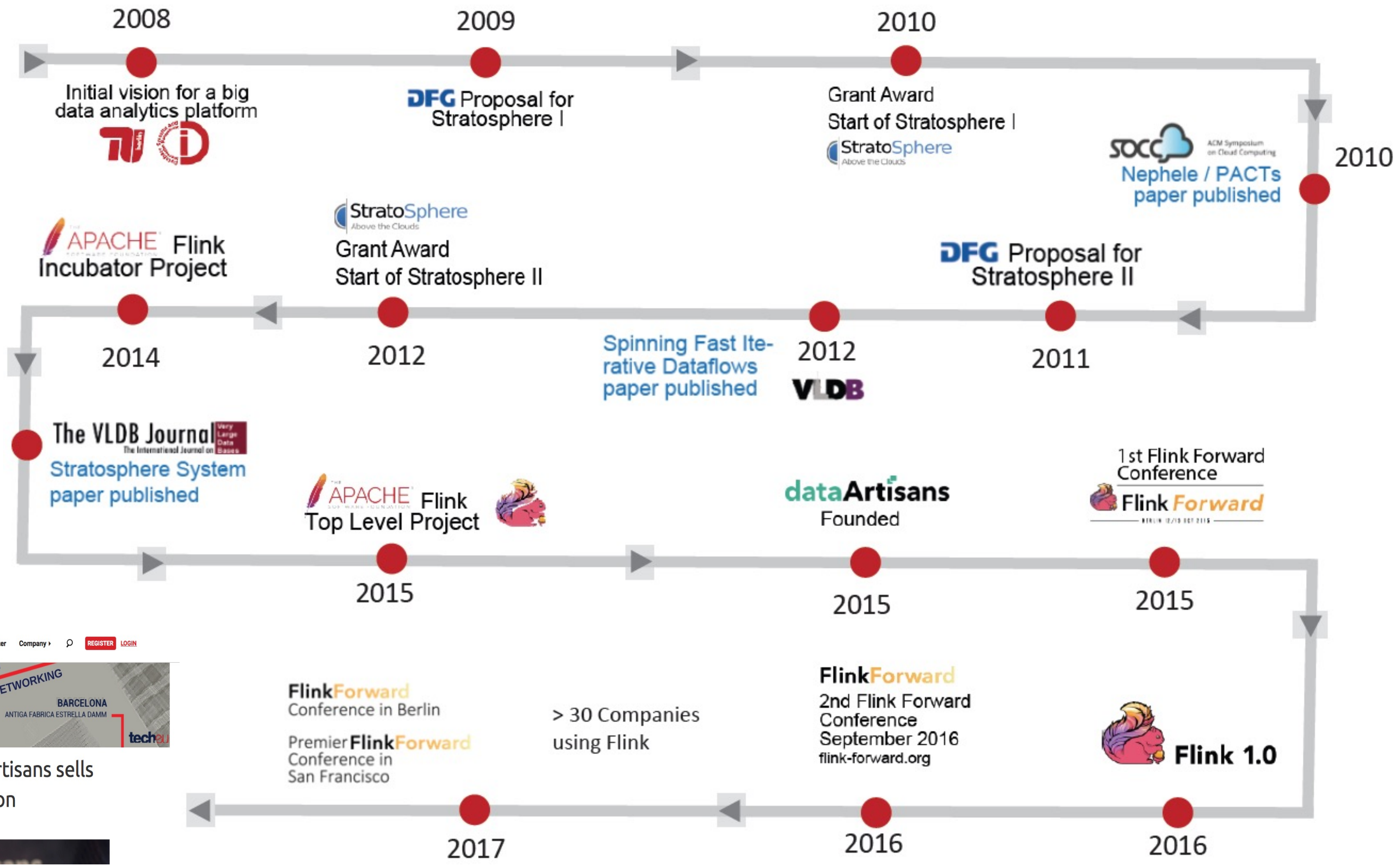


Visualization tools



Recapping the Evolution of Flink

Evolution Timeline of Flink

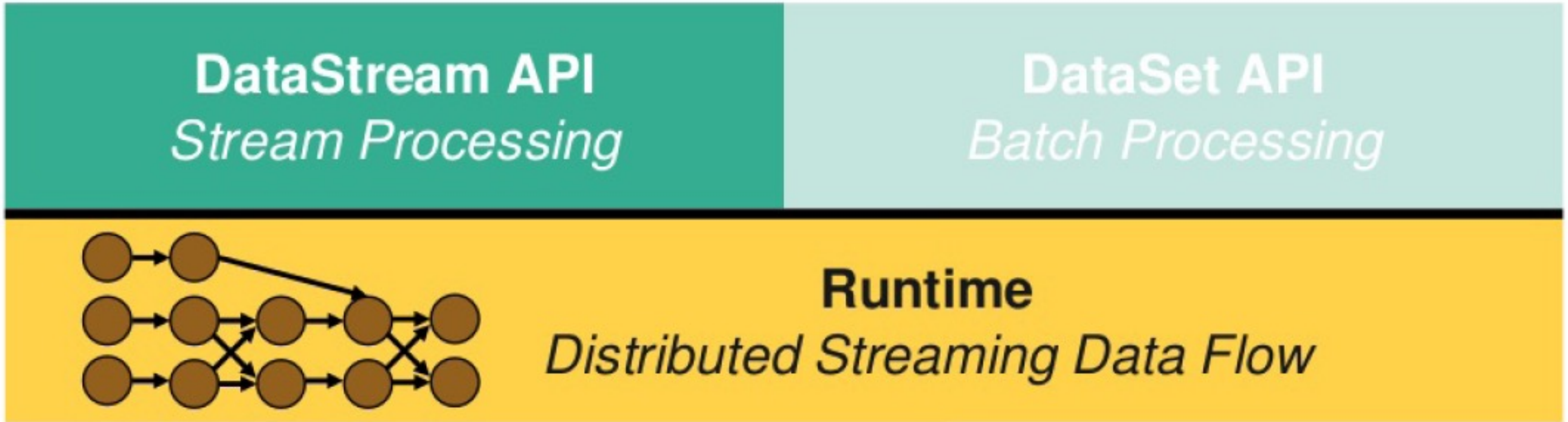


Evolution Timeline of Flink

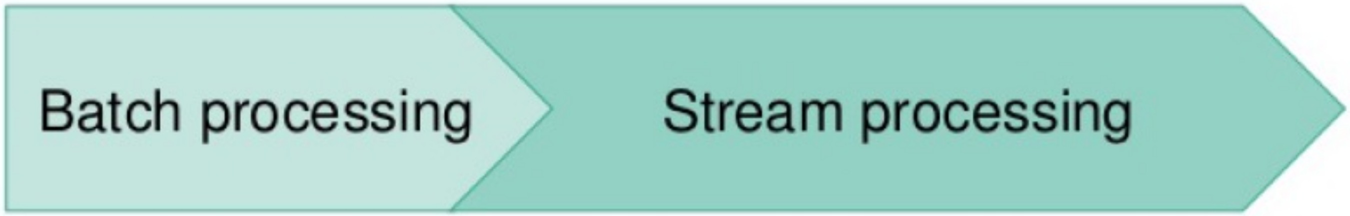
Batch processing

August 2014

Flink learns to stream in real time



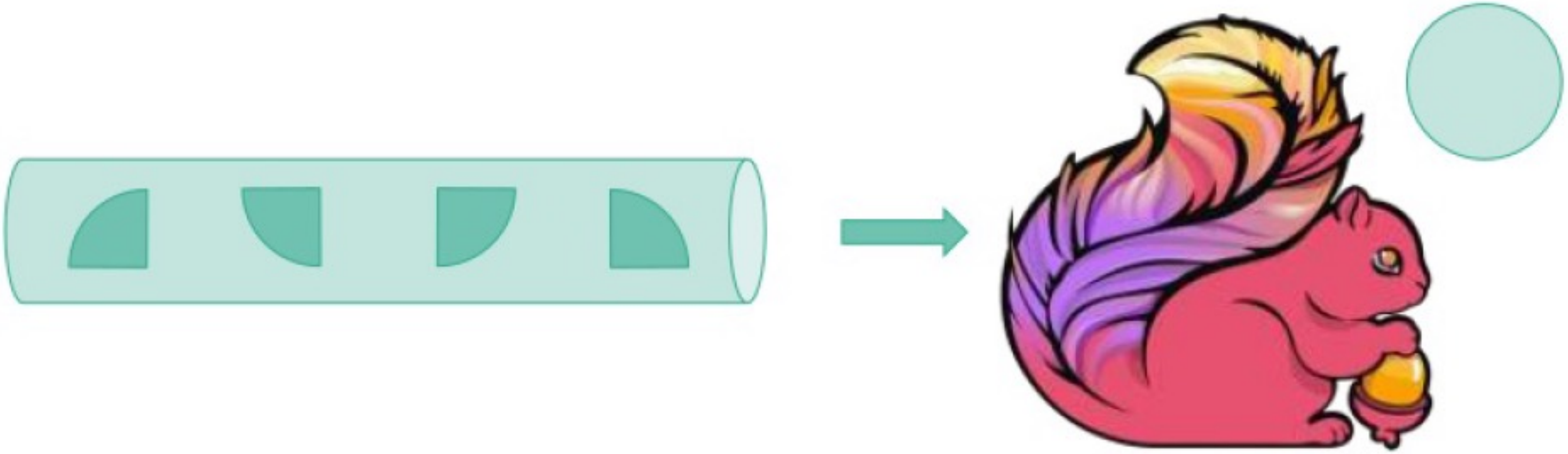
Evolution Timeline of Flink



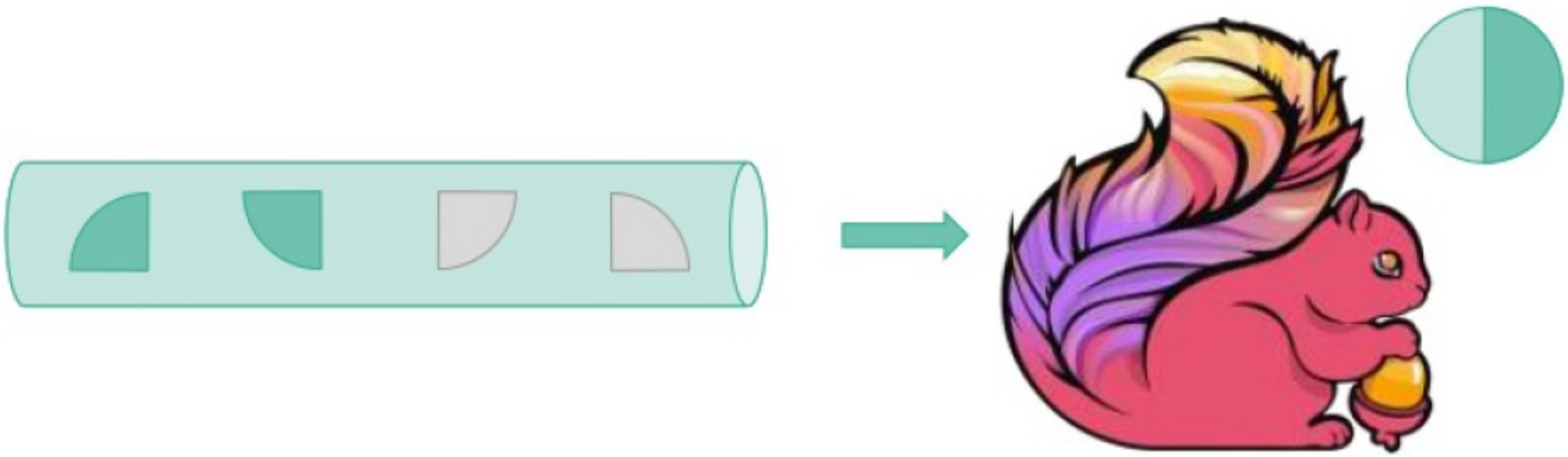
- Continuous & real-time

November 2014

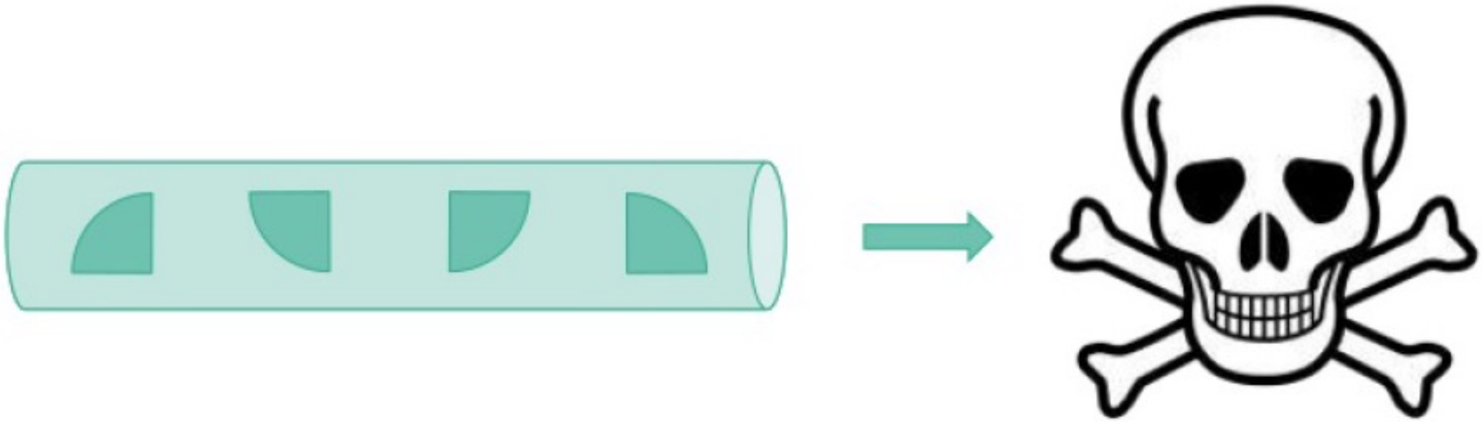
Flink learns to remember



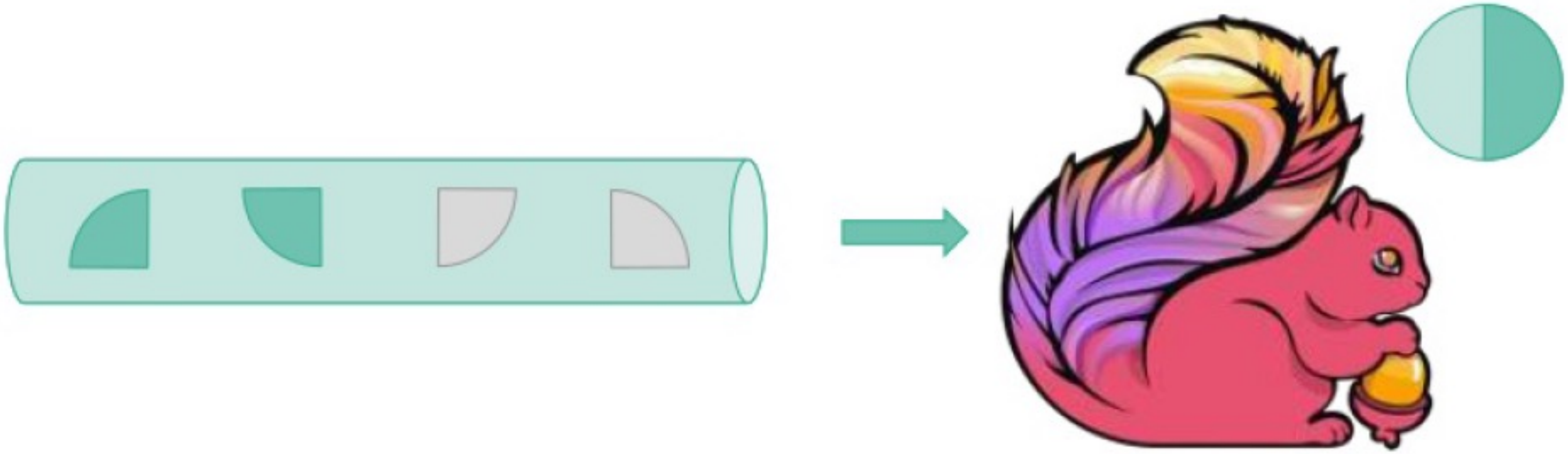
Flink learns to remember



Flink learns to remember

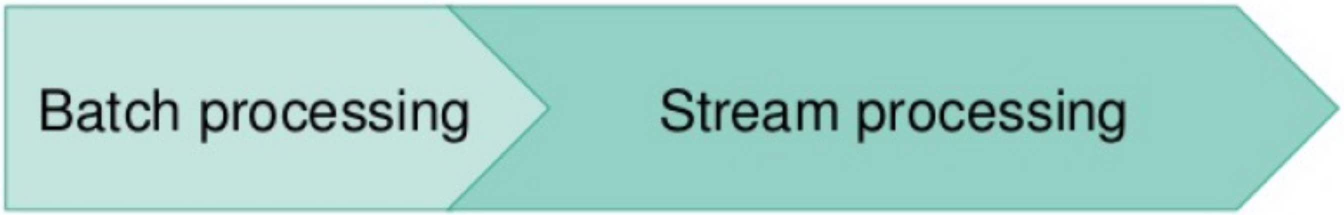


Flink learns to remember



Remember where we left off

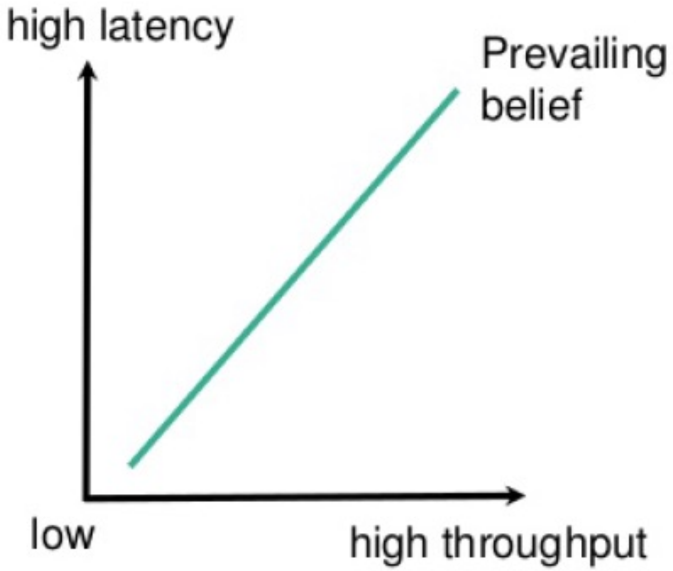
Evolution Timeline of Flink



- Continuous & real-time
- Stateful & exactly once

June 2015

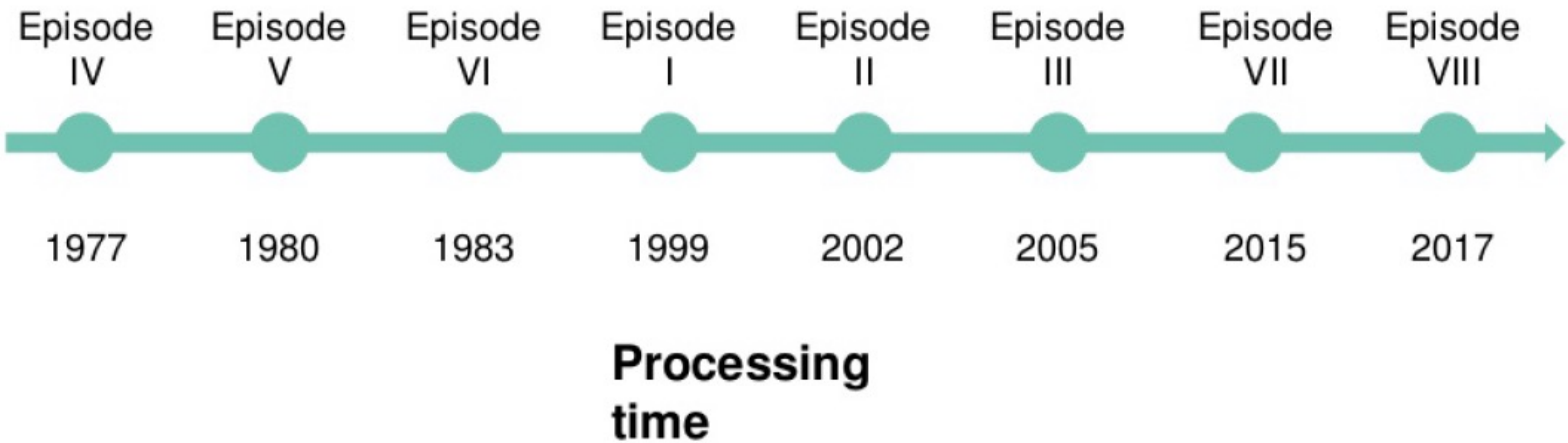
Latency vs. Throughput



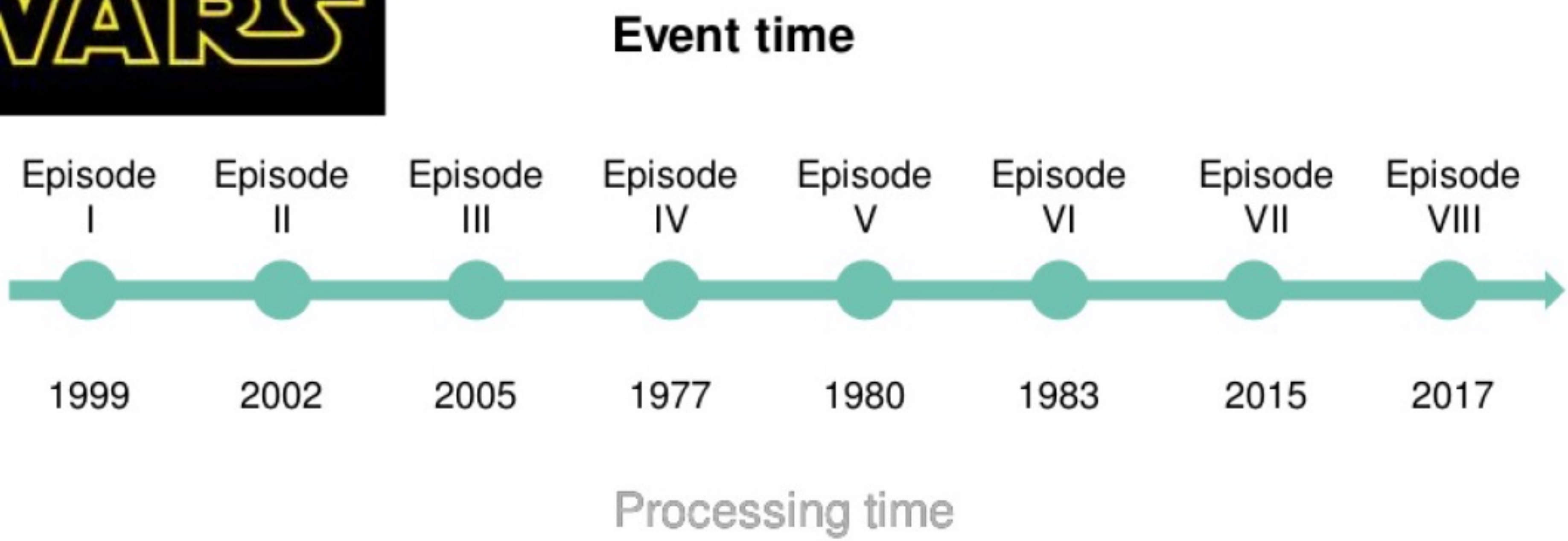
≠

- 10s of millions of events/s
- Latency down to 1 ms

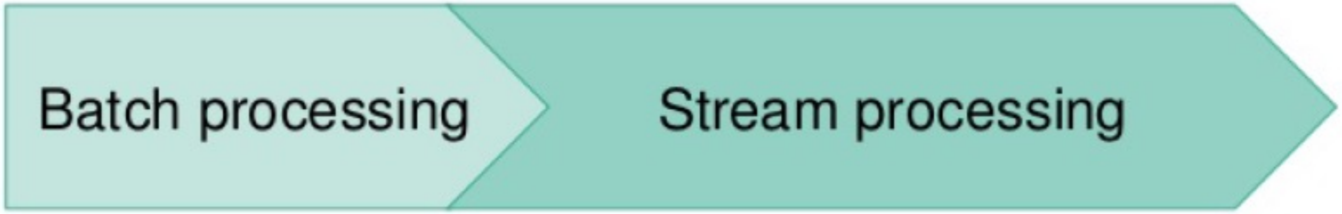
Flink becomes event-time aware



Flink becomes event-time aware



Evolution Timeline of Flink



- Continuous & real-time
- Stateful & exactly once
- High throughput & low latency
- Event time

November 2015

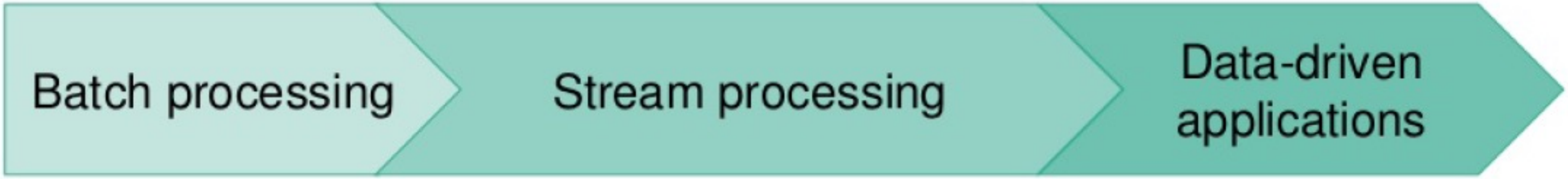
More than just analytics: ProcessFunction

```
class MyFunction extends ProcessFunction[MyEvent, Result] {

    // declare state to use in the program
    lazy val state: ValueState[CountWithTimestamp] = getRuntimeContext().getState(…)

    def processElement(event: MyEvent, ctx: Context, out: Collector[Result]): Unit = {
        // work with event and state and schedule timers
    }

    def onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[Result]): Unit = {
        // handle callback when event-/processing- time instant is reached
    }
}
```

- ProcessFunction gives access to state, time and events
- Low level API
- Enables data-driven applications

D_ TRB THE SOCIAL NETWORK FOR PETROLHEADS

Evolution Timeline of Flink



- Continuous & real-time
- Stateful & exactly once
- High throughput & low latency
- Event time

February 2017

Evolution Timeline of Flink (by v1.5)

- Continuous & real-time
- Stateful & exactly once
- High throughput & low latency
- Event time

- Applications as first class citizens

May 2018

Flink 1.5 in a nutshell

Hardening

- Faster network stack
- Application level flow control
- Resolving dependency hell

Scaling

- Incremental snapshots
- Local recovery
- Scalable timers

Interoperability

- Resource elasticity
- REST client-server interface
- Container entrypoint

Stream SQL

- SQL client
- User-defined functions
- More powerful joins

Misc

- State TTL
- Broadcast state
- Kafka exactly-once producer

New in Flink 1.5

- FLIP-6
  - Tighter integration with the resource manager (YARN, Mesos, Kubernetes)
  - Enables dynamic management of resources
  - Rework of the client/cluster communication to be REST-based
- Localised Failure Recovery
  - Failures don't require restoring all state from distributed storage
  - TaskManagers keep state on machines
  - Failures that are not caused by machine failures lead to faster recovery
- 50% Network Stack Rewrite
  - Better throughput at very low latencies
  - Much improved backpressure handling

New in Flink 1.5 (cont'd)

- Broadcast State
  - API that enables new use cases such as applying dynamic CEP patterns on a stream or join
- SQL CLI
  - An interactive command-line interface for executing SQL queries on Flink
- Unified Table Sources
  - A new interface for defining sources for a Table API/SQL program that allows defining sources from a configuration file
- Loads more automated testing/release verification
  - Streamlined testing which will lead to lower overhead for releases

Flink 1.6 and Beyond

v1.6 released in Aug 2018,
v1.7 in Nov 2018

What’s new in Flink 1.6

- Autoscaling
  - Automatic and dynamic changes in the parallelism of Flink programs and individual operators
- Hot-standby replication
  - Replication of the state of operations to multiple machines so that we can instantly migrate computation in case of failures
- Zero-downtime scaling and upgrades
  - Parallelism changes, framework upgrades and user-code updates without any downtime

What’s new in Flink 1.6 (cont’d)

- More Table API/SQL connectors, integration with data bases
  - Dynamic Tables based on a data base, not a stream
- End-to-end batch/streaming integration
  - Unification of the DataStream and DataSet APIs
  - Efficient execution of batch programs and streaming programs
  - Dynamic switching of execution modes based on workload
- Support for more programming languages
  - Upcoming: Python and Go (via Apache Beam)
  - Tensorflow for Machine Learning and AI (also via Apache Beam)

What's new in Flink 1.6 (cont'd)

- Java 9 (FLINK-8033) and Scala 2.12 (FLINK-7811)
- Improvements for container environments, e.g. K8s (FLINK-9495)
- Full job submission through REST (FLINK-9280)
- State back-ends for timers (FLINK-9485)
- State back-ends for operator state

What’s new in Flink 1.6 (cont’d)

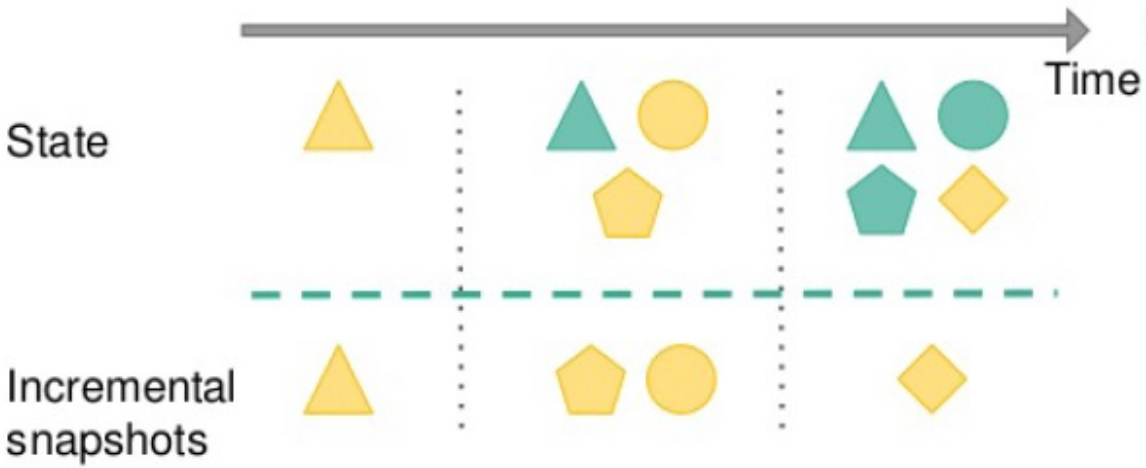
- BucketingSink with Flink file systems (including S3)
- State evolution: support type conversion on snapshot restore
- Stream SQL:
  - support “update by key” Table Sources
  - more table sources and sinks (Kafka, Kinesis, Files, K/V stores)
- CEP
  - Integrate CEP and SQL via MATCH_RECOGNIZE (FLINK-7062)
  - Improve CEP performance of SharedBuffer on RocksDB (FLINK-9418)

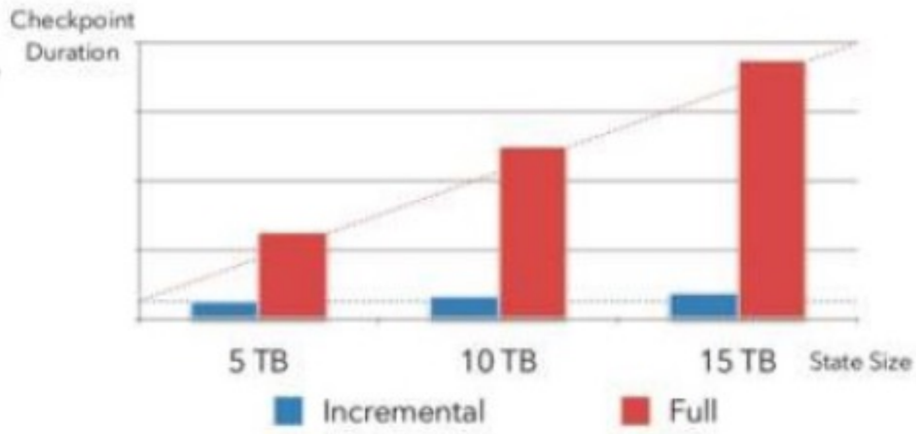
Major New Features in Flink 1.7

- Support of State (Schema) Evolution
- Exactly-Once support with AWS S3-streaming
- MATCH_RECOGNIZE support in Streaming SQL
- Temporal Tables and Temporal Joins in Streaming SQL

More Details on New Features in Flink 1.5 and Beyond

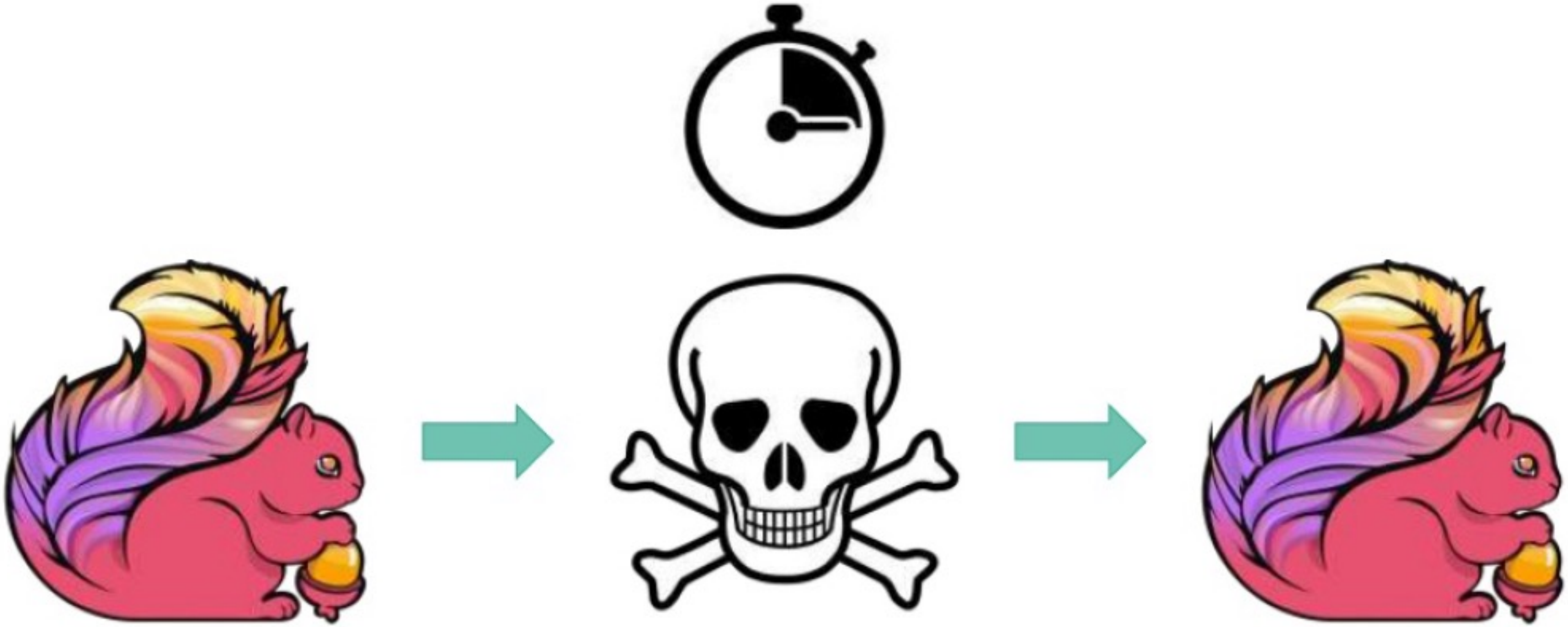
Large, larger, Flink





- Snapshot only state diff
- Incremental snapshots allow to handle very large state

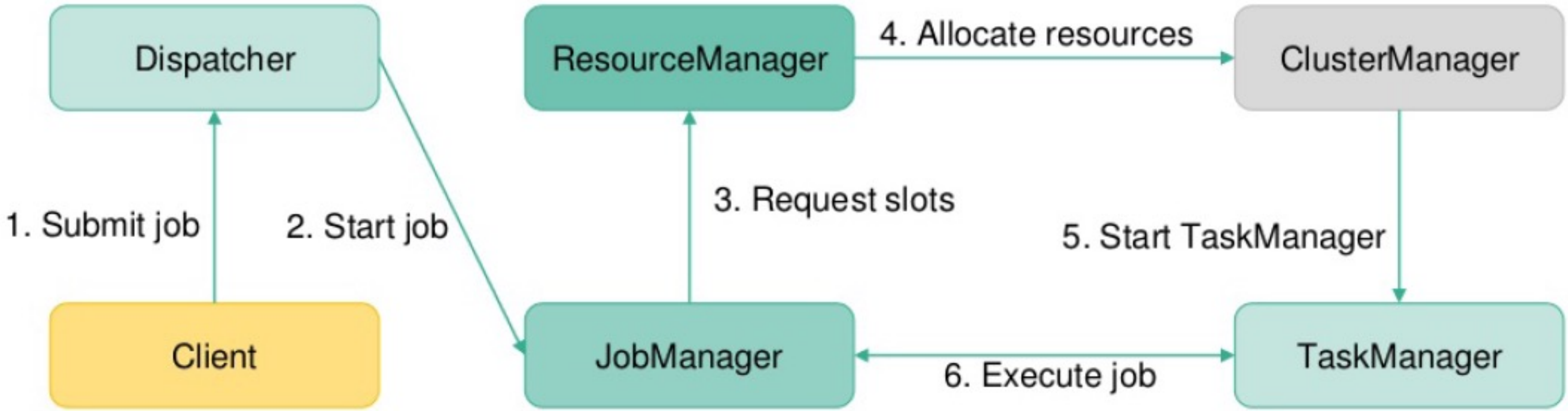
Faster failover is always better



Varying workloads

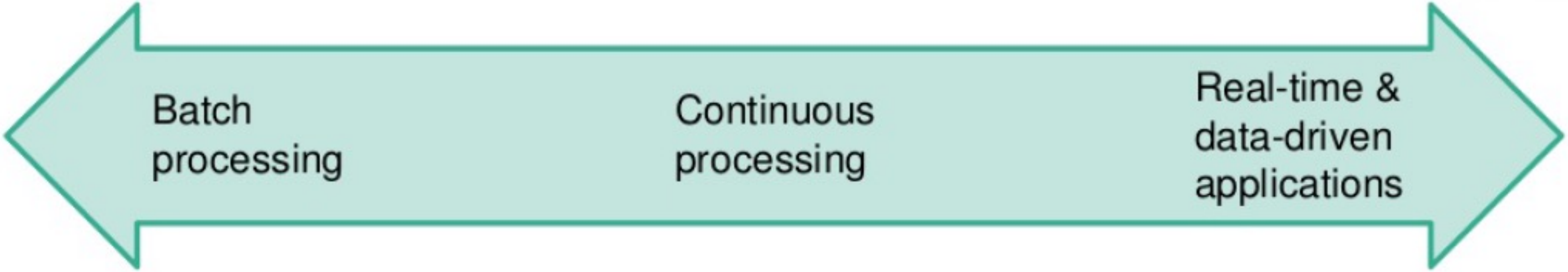
- Violating SLAs vs. wasting money
- Varying workloads require to adapt resources

Revamped distributed architecture



- Support for full resource elasticity
- Application parallelism can be dynamically changed

How much control do I need?

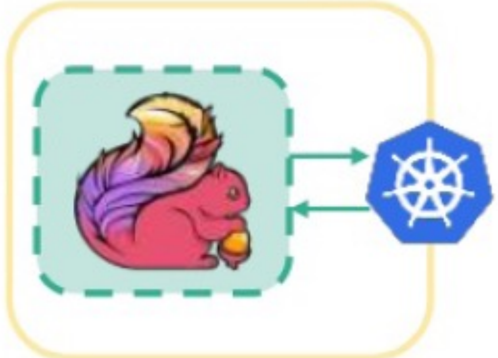


- Multiple short lived stages
- Different resource requirements per stage
- Efficient execution requires control over resources
- Flink allocates actively resources

- Continuously processing operators
- Constrained by external systems, SLAs and application logic
- External system can assign resources
- Flink reacts to available resources

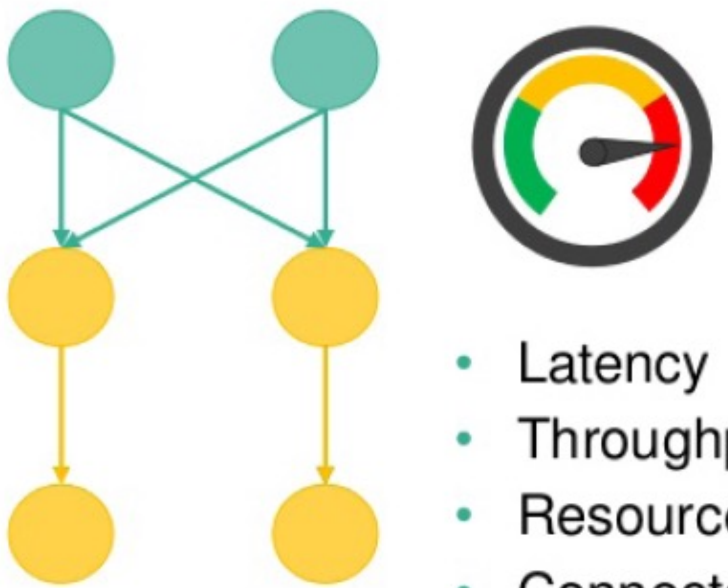
Active vs. reactive mode

- Active mode
  - Flink is aware of underlying cluster framework
  - Flink allocate resources
  - E.g. existing YARN and Mesos integration

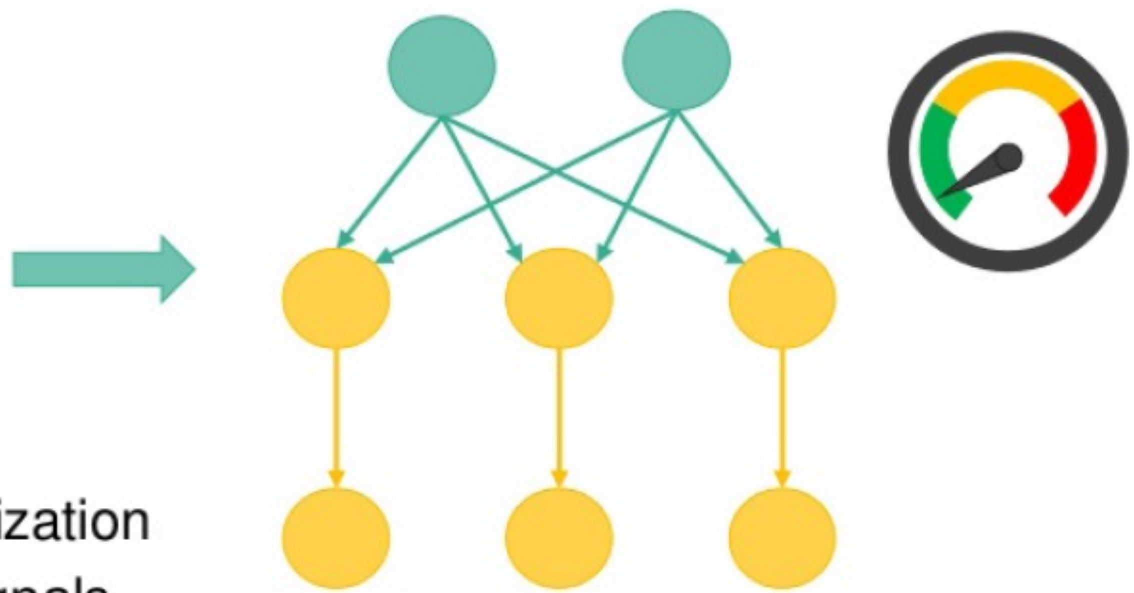


- Reactive mode
  - Flink is oblivious to its runtime environment
  - External system allocates and releases resources
  - Flink scales with respect to available resources
  - Relevant for environments: Kubernetes, Docker, as a library

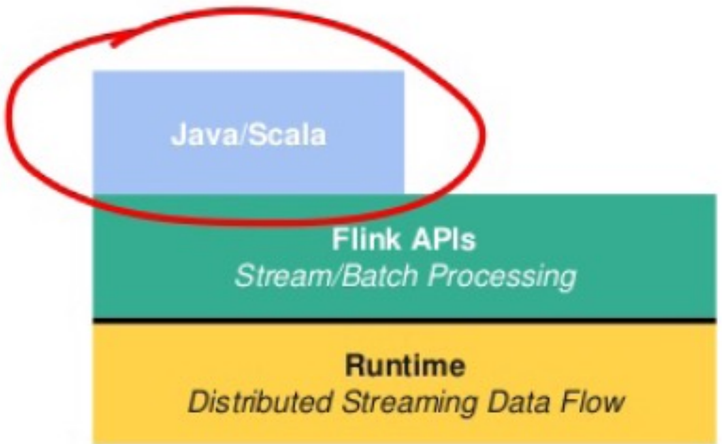
Scaling automatically

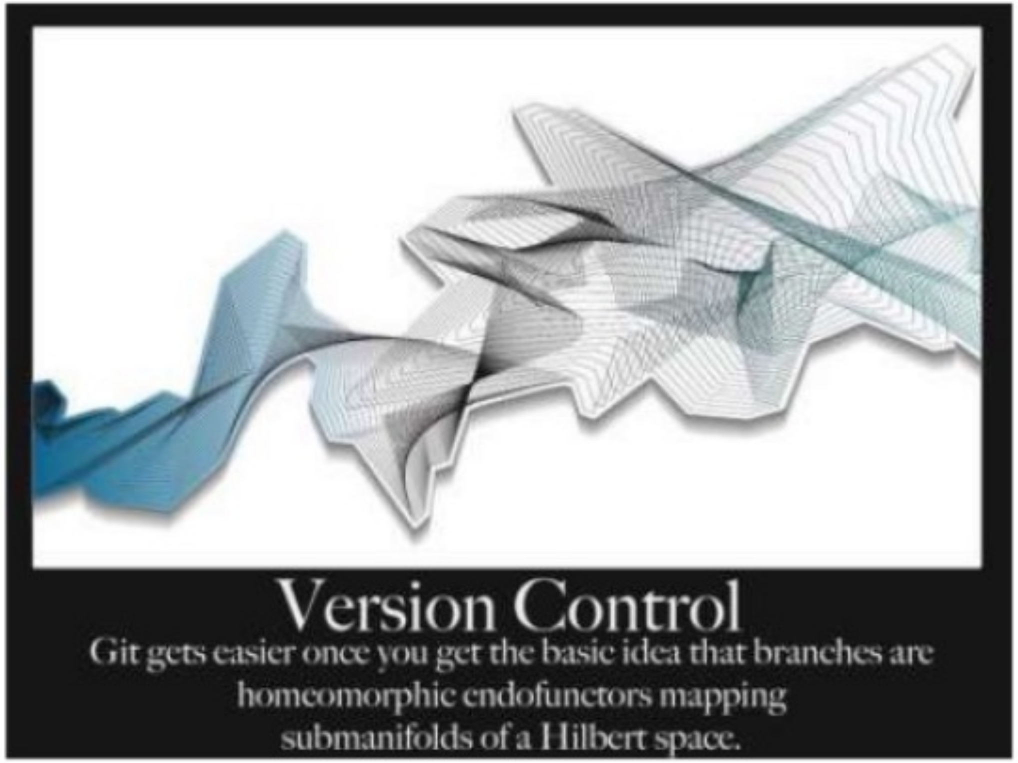


- Latency
- Throughput
- Resource utilization
- Connector signals



How we create Flink Jobs





Powered by Apache Flink

Alibaba Group · ING · NETFLIX · UBER · HUAWEI · COMCAST

King · CISCO · accenture · Expedia · ebay · Capital One

DELL EMC · yelp · lyft · here · DTRB · zalando

MediaMath · amadeus · LINE · Telefónica · bouygues telecom · Lightbend

bol.com · SK telecom · OVH.com · relayr. · BetterCloud · otto group

Retail, e-commerce

- Better product recommendations
- Process monitoring
- Inventory management

Finance

- Differentiation via tech
- Push-based products
- Fraud detection

Telco, IoT, Infrastructure

- Infrastructure monitoring
- Anomaly detection

Internet & mobile

- Personalization
- User behavior monitoring
- Analytics

Flink in Practice (by Sept 2016)

Largest job has > 20 operators, runs on > 5000 vCores in 1000-node cluster, processes millions of events per second

Complex jobs of > 30 operators running 24/7, processing 30 billion events daily, maintaining state of 100s of GB with exactly-once guarantees

30 Flink applications in production for more than one year. 10 billion events (2TB) processed daily

Flink in Practice: more sample applications

UBER

Athena X Streaming SQL
Platform Service

NETFLIX

Streaming Platform as a Service

3700+ Docker containers running Flink
1400+ nodes with 22K+ cpu cores

4000+ Kafka brokers, 50+ clusters
100's of Data Streams (Flink Jobs)

Alibaba Group

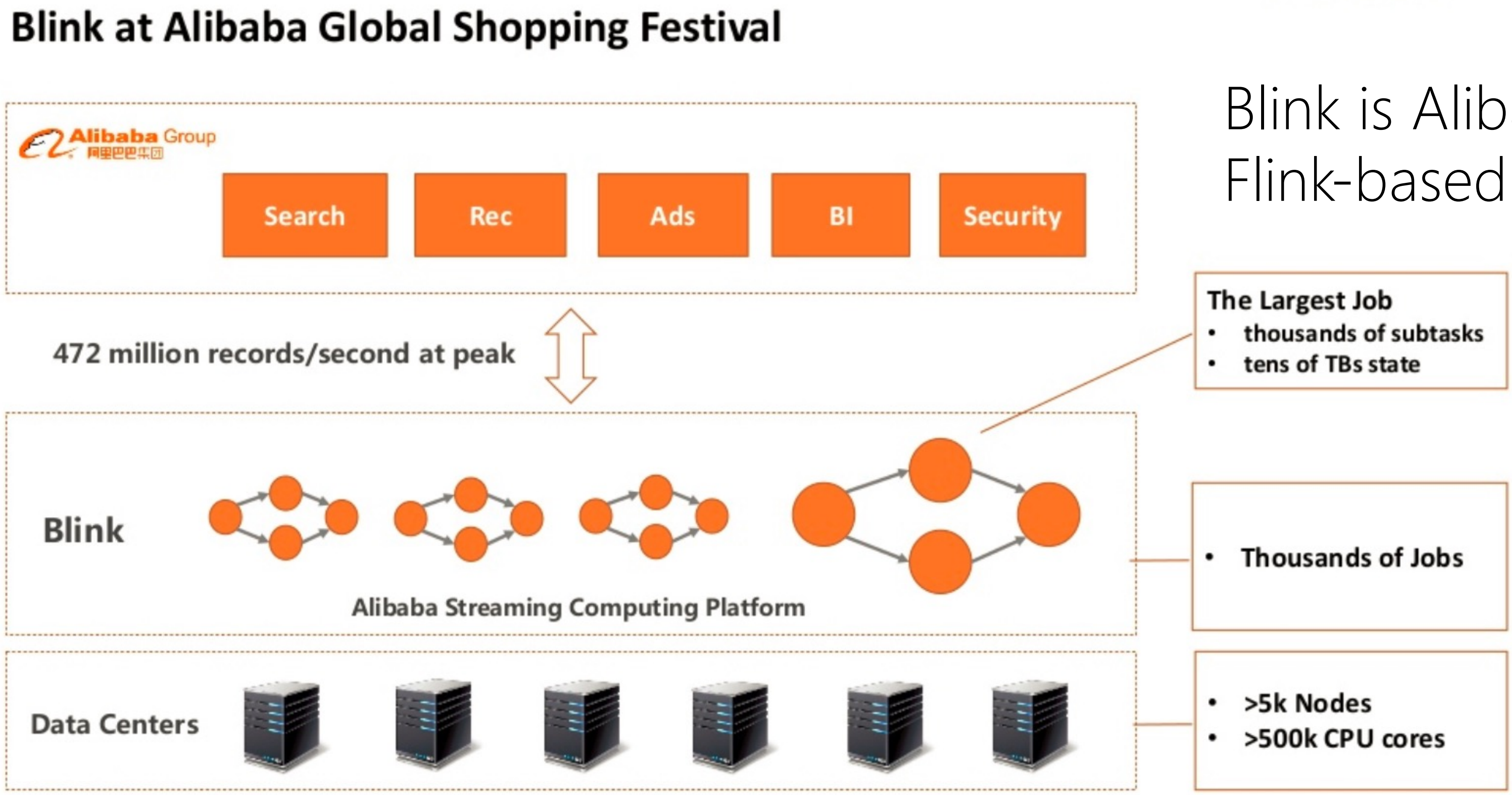
100s jobs, 1000s nodes, TBs state
metrics, analytics, real time ML
Streaming SQL as a platform

ING

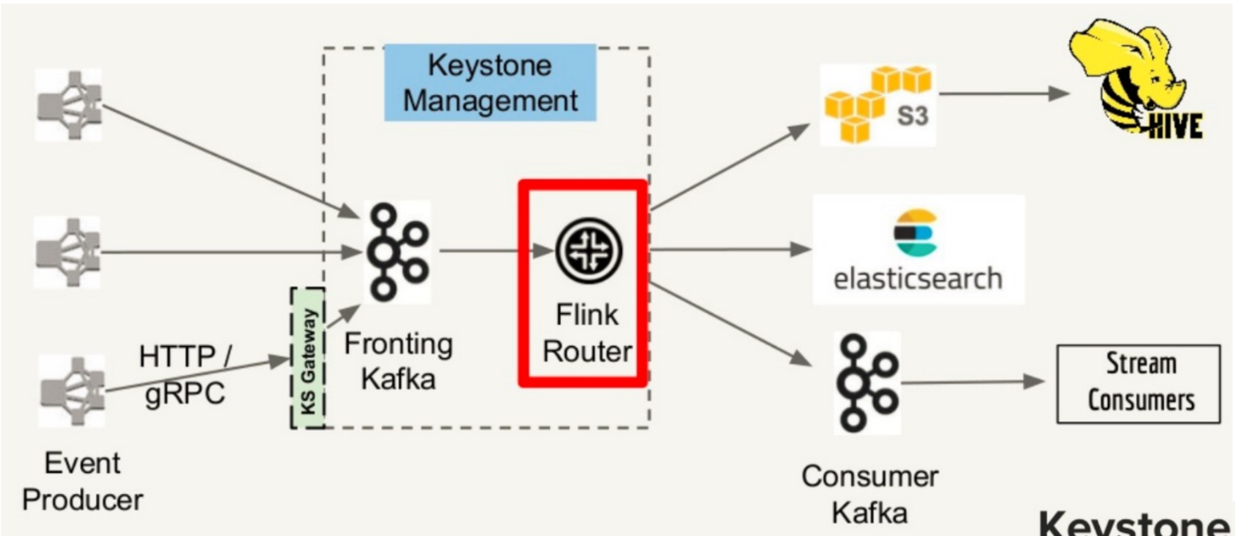
Fraud detection
Streaming Analytics Platform

How Large (or Small) can Flink get?

Blink is Alibaba's Flink-based System



Keystone Routing Pipeline at Netflix
(as presented at Flink Forward San Francisco, 2018)



Keystone router scale

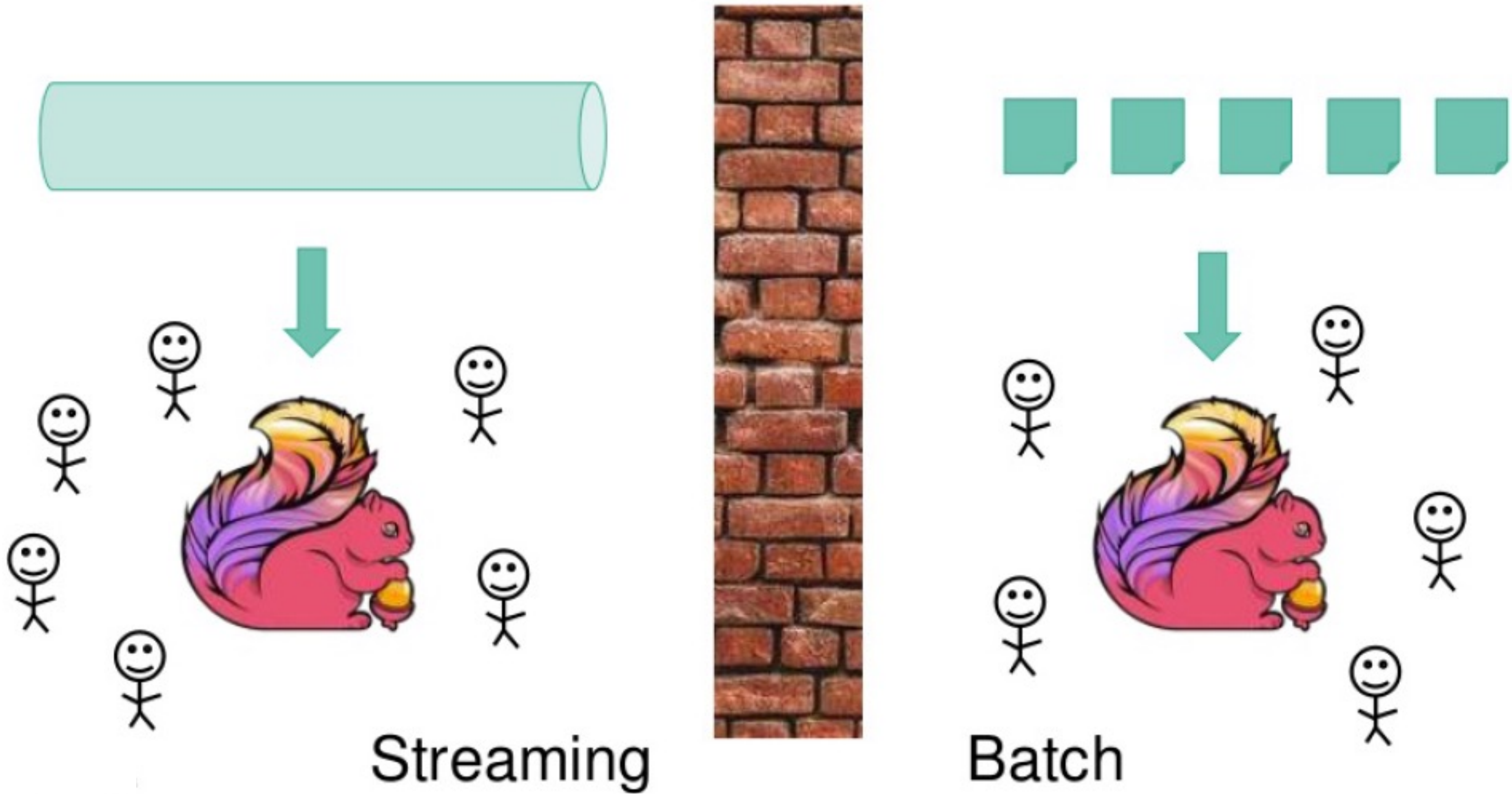
- ~3 trillion events/day
- ~2,000 routing jobs
- ~10,000 containers
- ~200,000 parallel operator instances

Small Flink

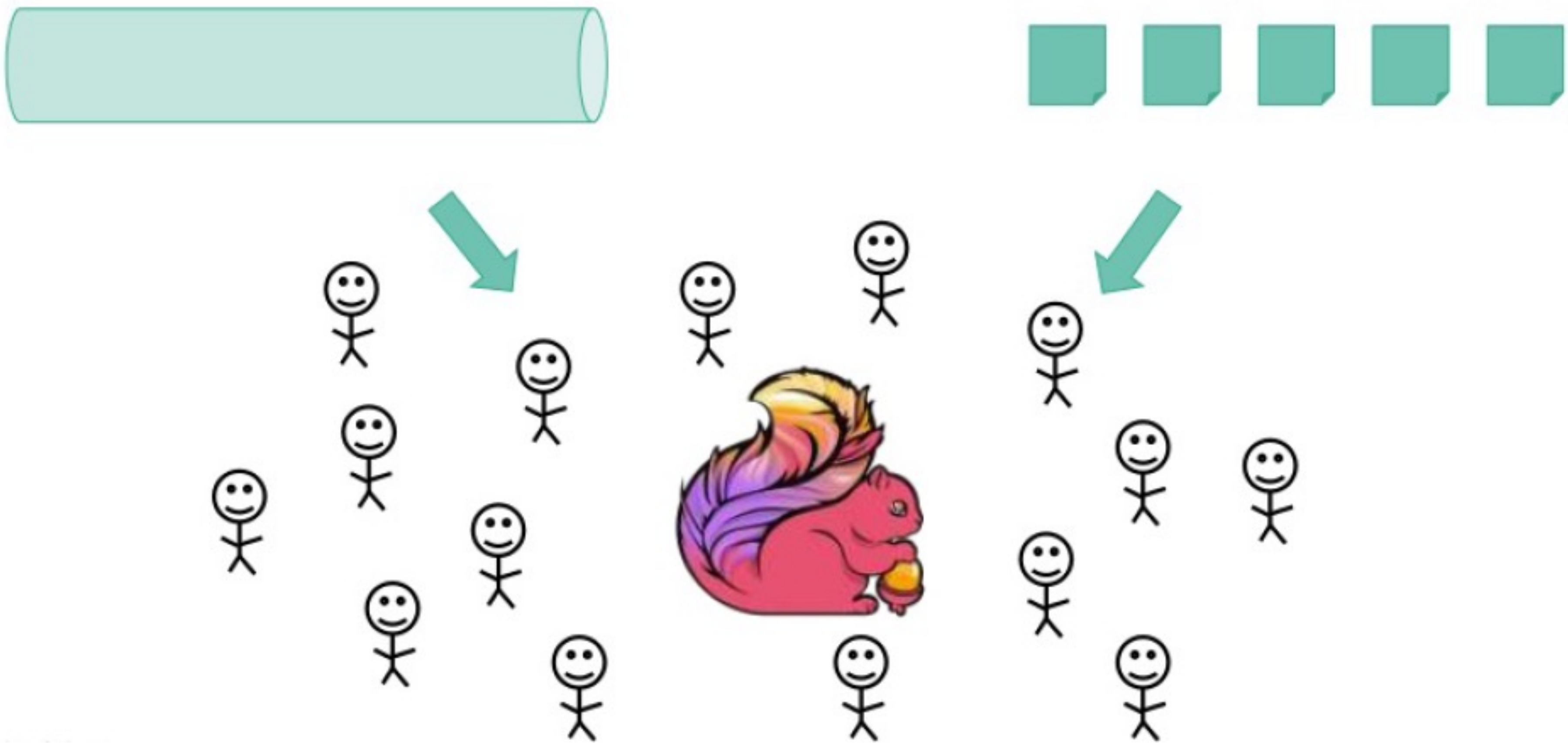
- Can run in single process
- Some users run it on IoT Gateways
- Also runs with zero dependencies in IDE

Future Direction for Flink

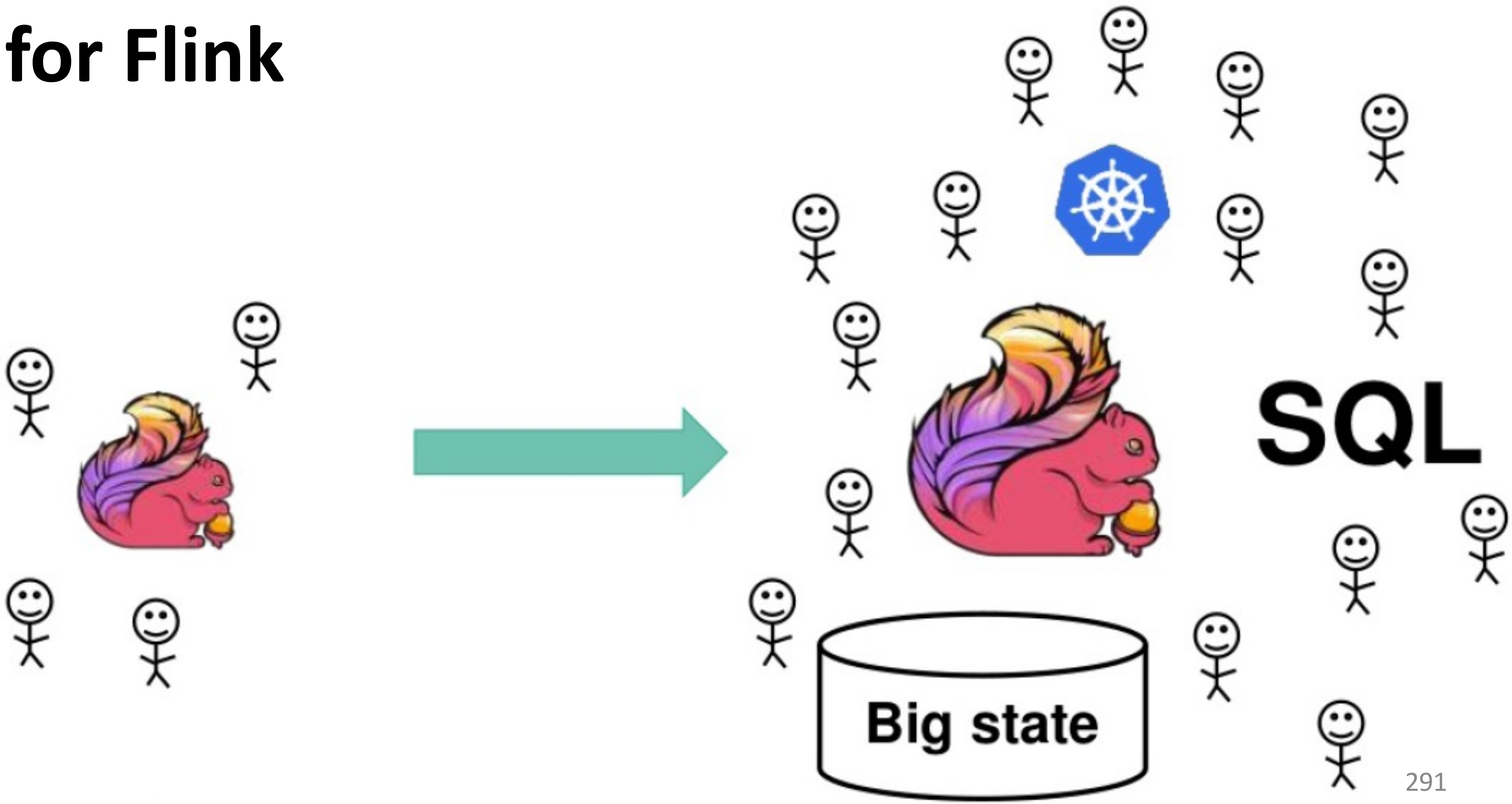
Todays processing landscape



What's Next: True Batch/ Stream Unification



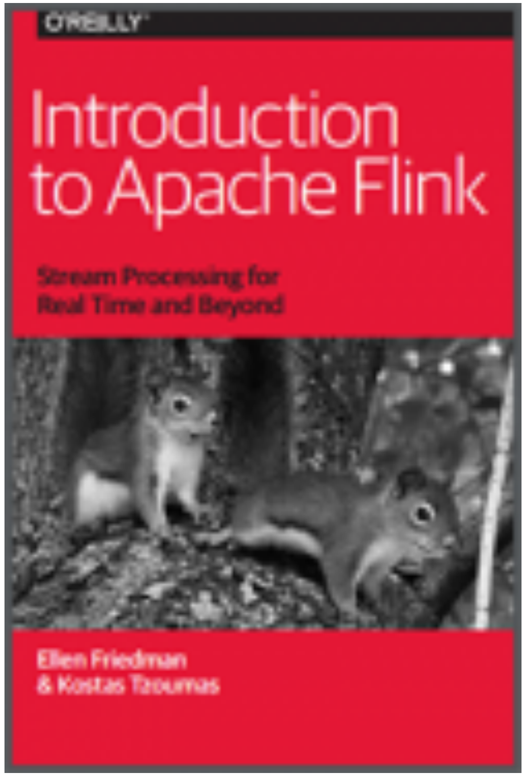
Other Ongoing Objectives for Flink

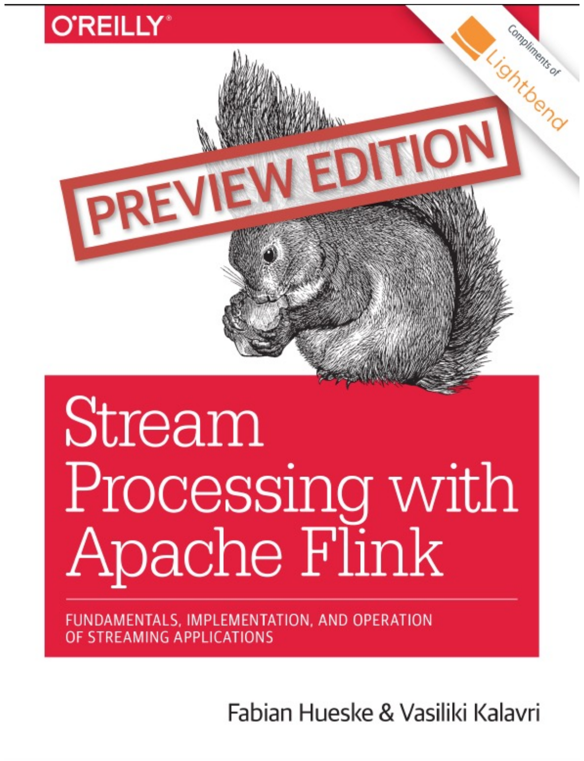


Other Ongoing Objectives for Flink

- Provide state of the art streaming capabilities
- Operate in the largest infrastructures of the world
- Open up to a wider set of enterprise users
- Broaden the scope of stream processing

Authoritative Free Books on Apache Flink



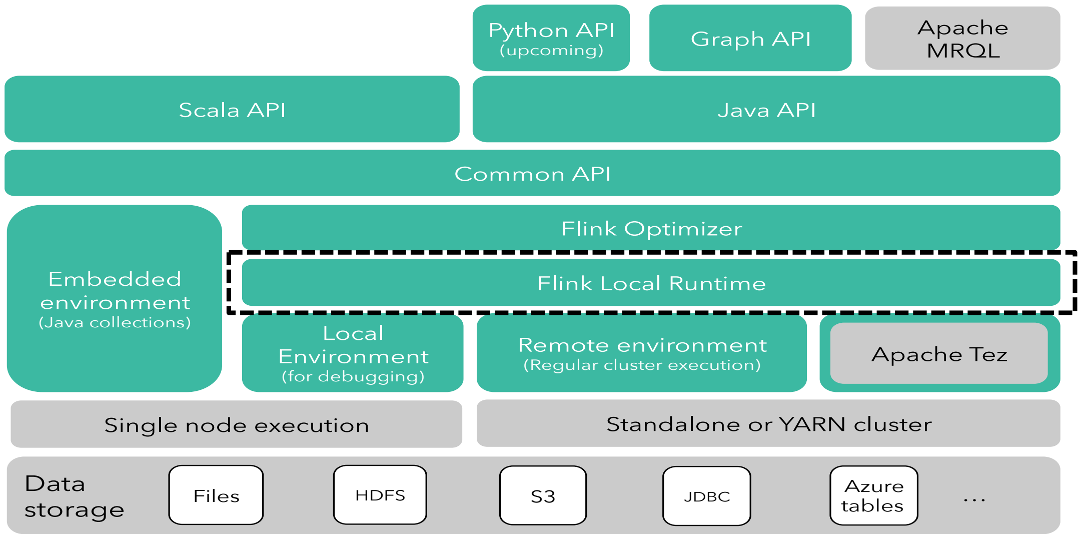


Available at:
https://mapr.com/introduction-to-apache-flink/
https://info.lightbend.com/rs/558-NCX-702/images/preview-apache-flink.pdf

Backup Slides

Flink runtime features

Flink Local Runtime



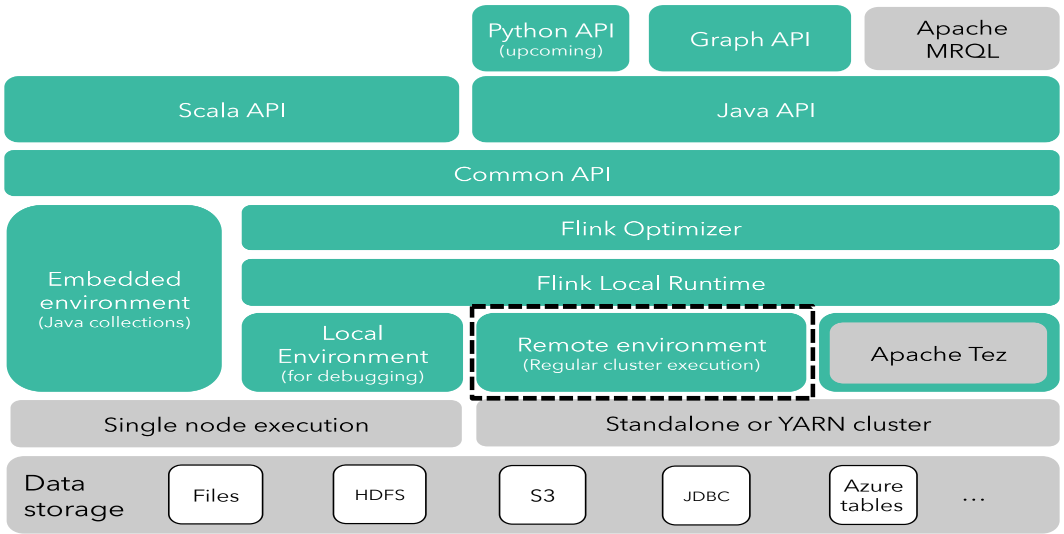
- *Local* runtime, not the distributed execution engine
- Aka: what happens inside every parallel task

Flink runtime operators

- Sorting and hashing data
  - Necessary for grouping, aggregation, reduce, join, cogroup, delta iterations

- Flink contains tailored implementations of hybrid hashing and external sorting in Java
  - Scale well with both abundant and restricted memory sizes

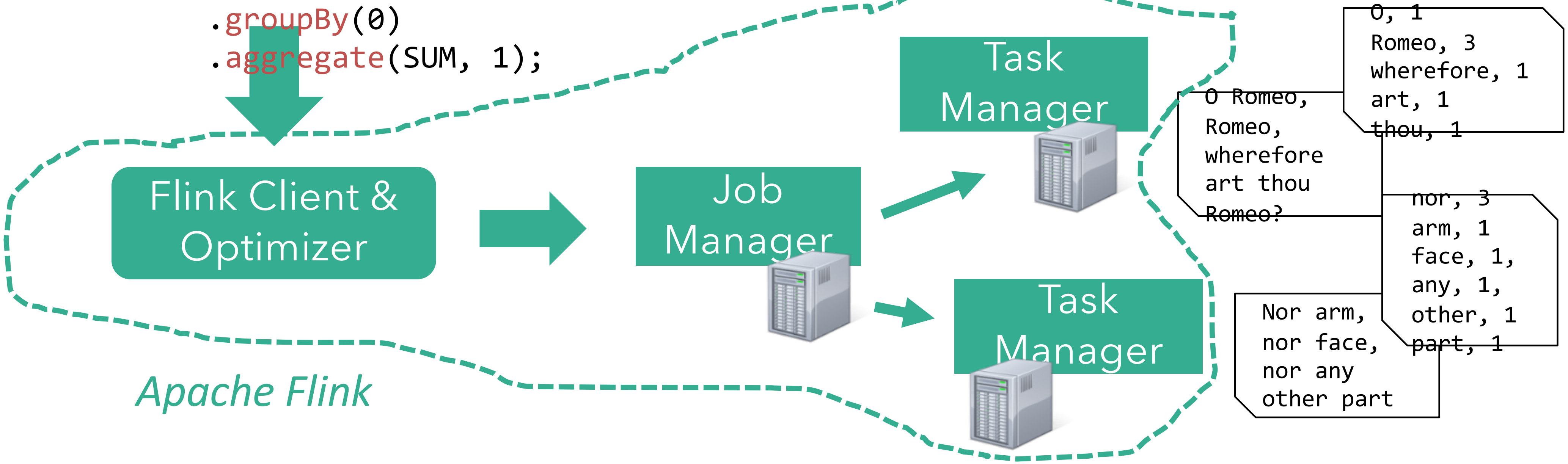
Flink distributed execution



Coordination built on Akka library

- Pipelined
  - Same engine for Flink and Flink streaming
- Pluggable
  - Local runtime can be executed on other engines
  - E.g., Java collections and Apache Tez

```java
DataSet<String> text = env.readTextFile(input);

DataSet<Tuple2<String, Integer>> result = text
        .flatMap((str, out) -> {
            for (String token : value.split("\\W")) {
                out.collect(new Tuple2<>(token, 1));
            })
        .groupBy(0)
        .aggregate(SUM, 1);
```



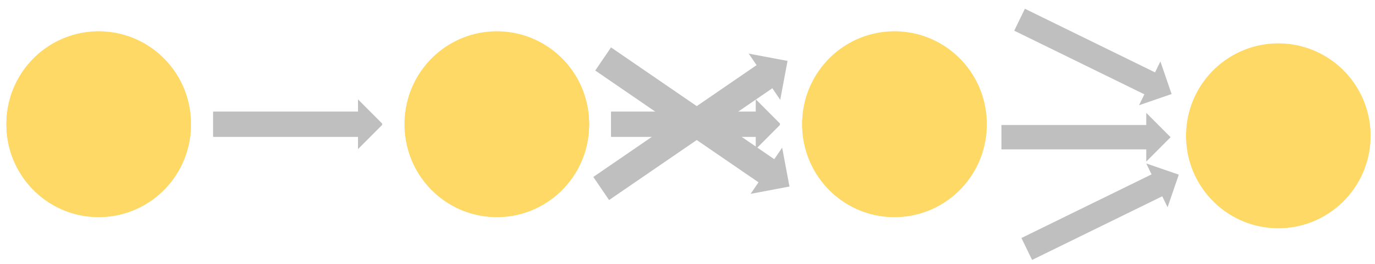
If you need to know **one** thing about Flink is that you don’t need to know the internals of Flink.

Philosophy

- Flink “hides” its internal workings from the user
- This is **good**
  - User does not worry about how jobs are executed
  - Internals can be changed without breaking changes
- … and **bad**
  - Execution model more complicated to explain compared to MapReduce or Spark RDD

Parallel Stateful Streaming Execution

Stateful Event & Stream Processing



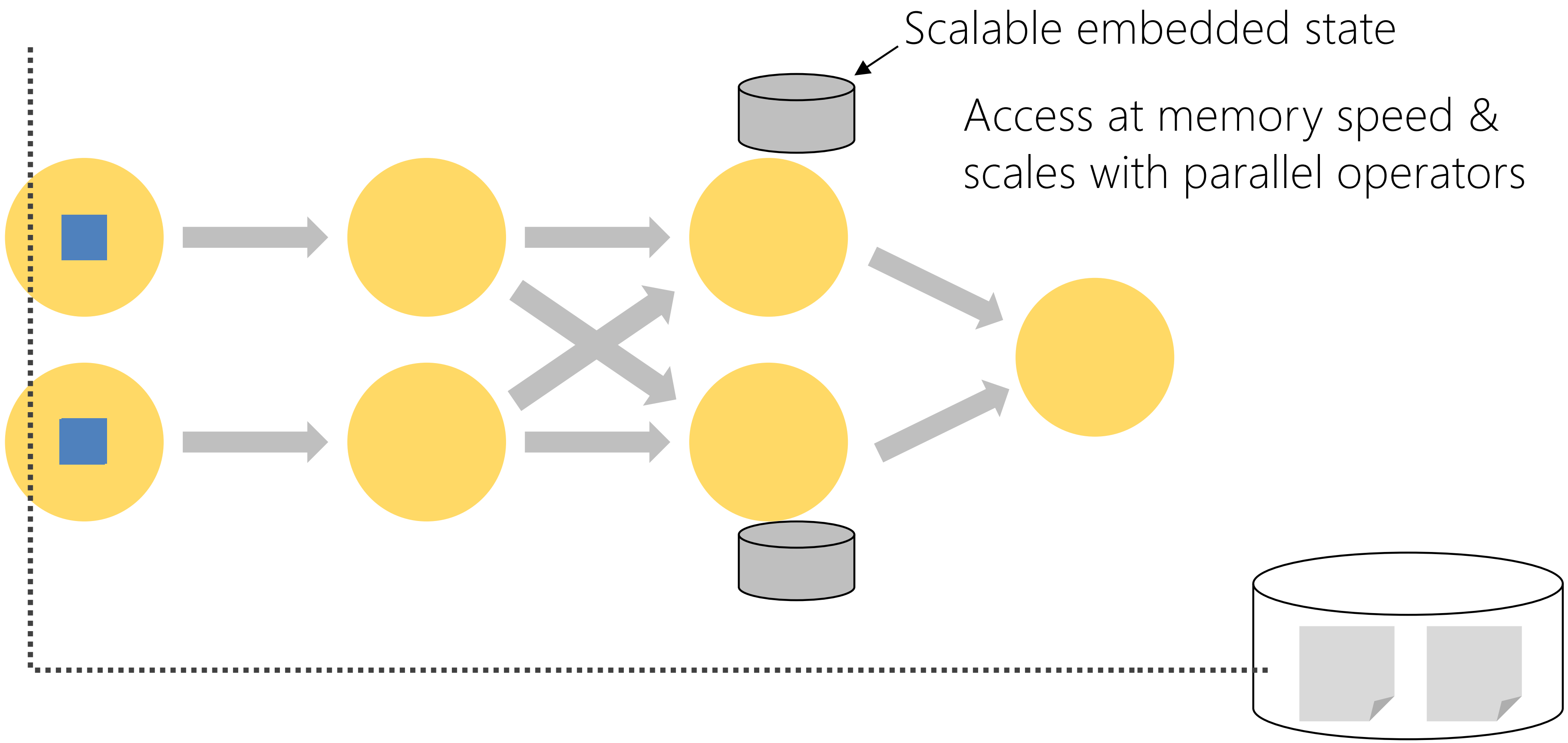
Source

Filter / Transform

State read/write

Sink

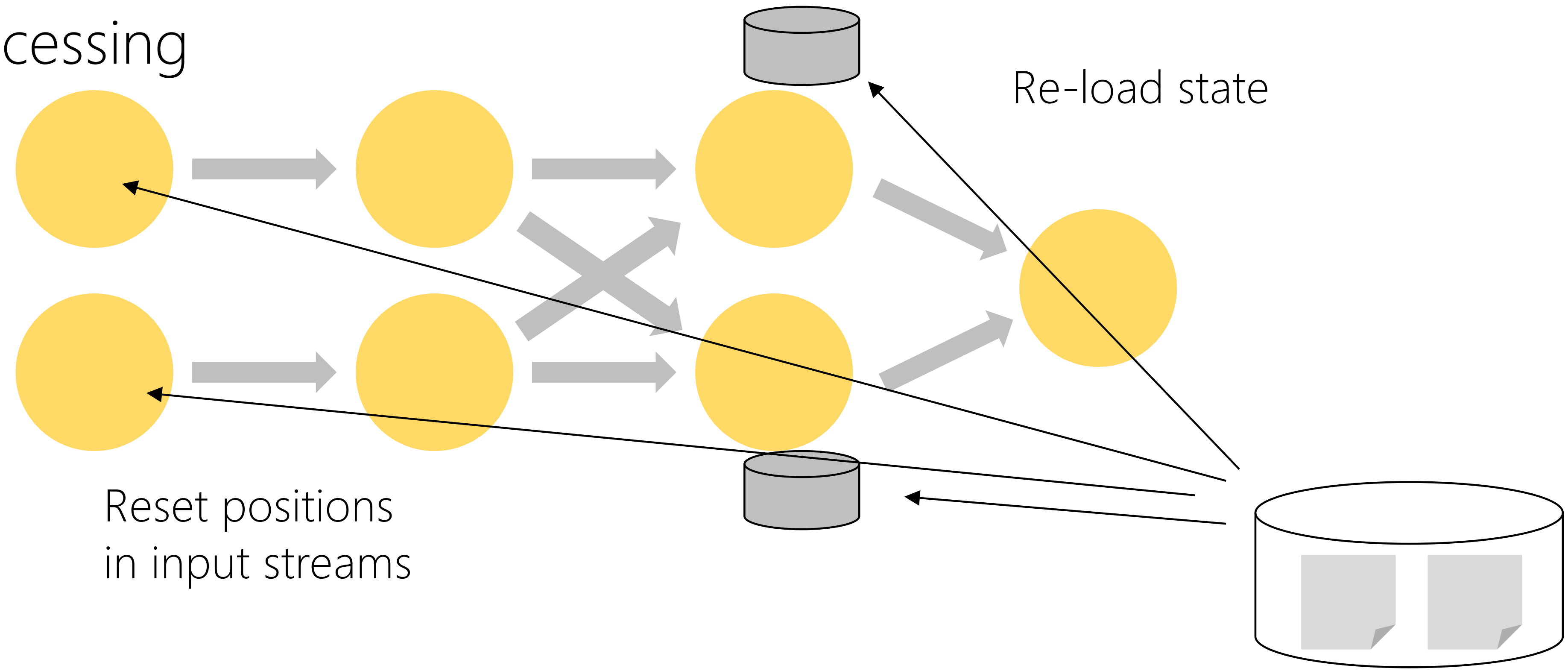
Stateful Event & Stream Processing



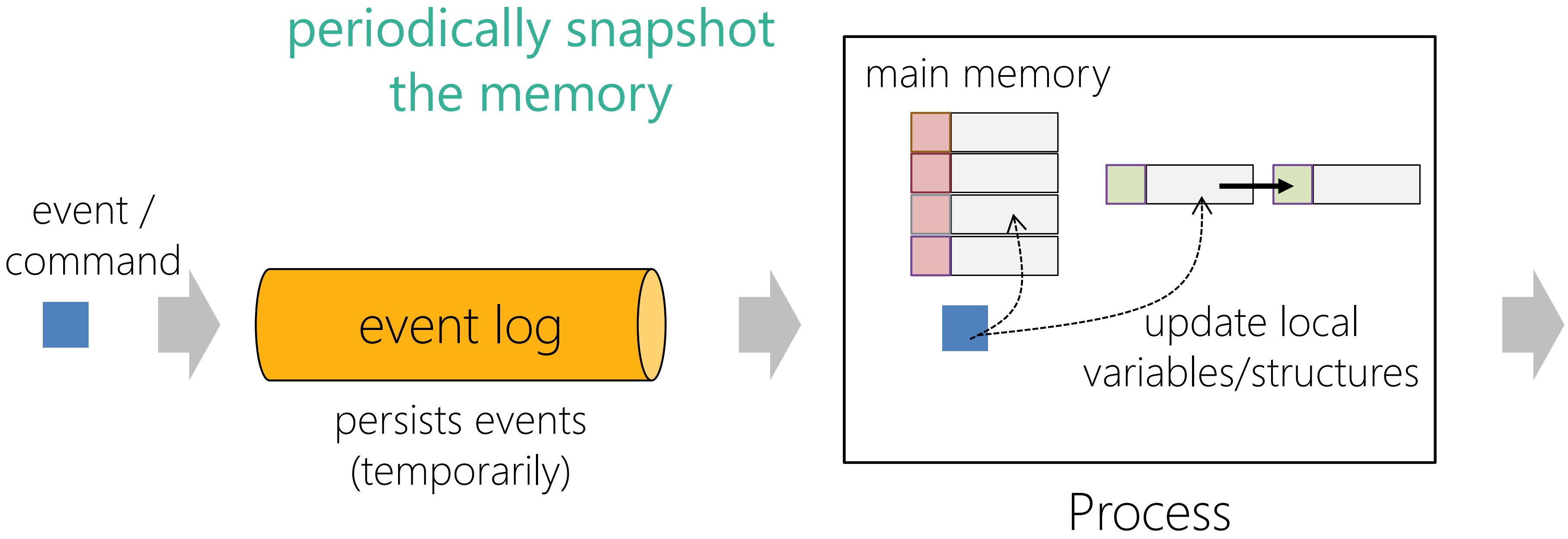
Stateful Event & Stream Processing

Rolling back computation

Re-processing

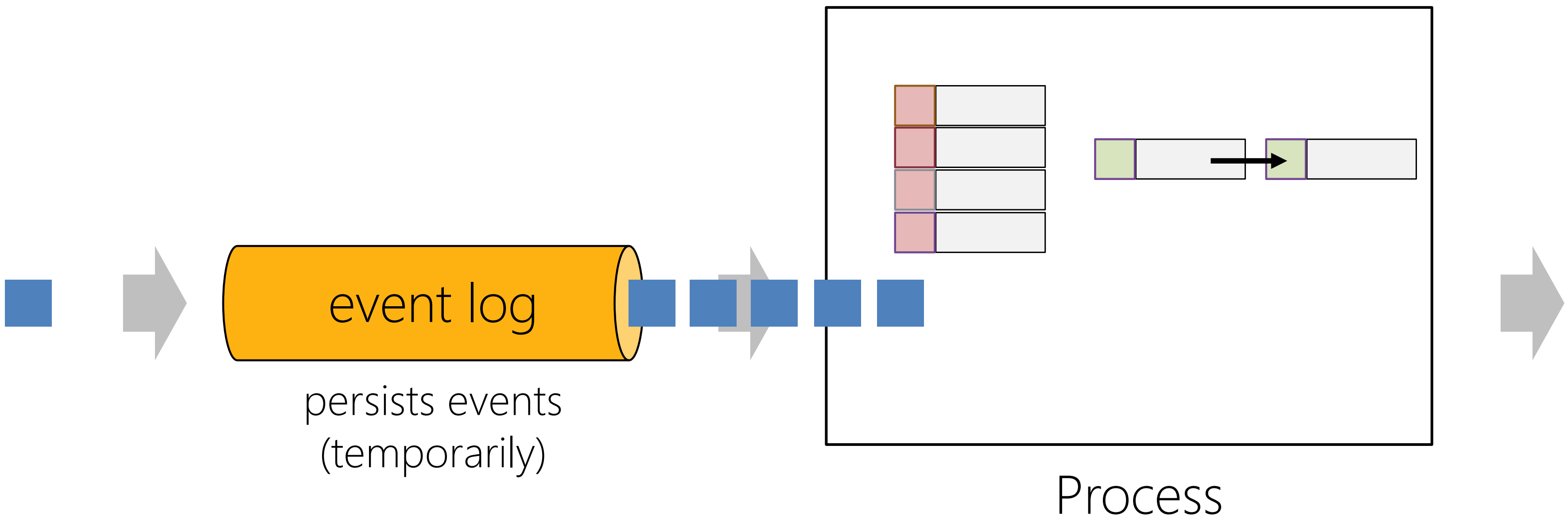


Event Sourcing + Memory Image

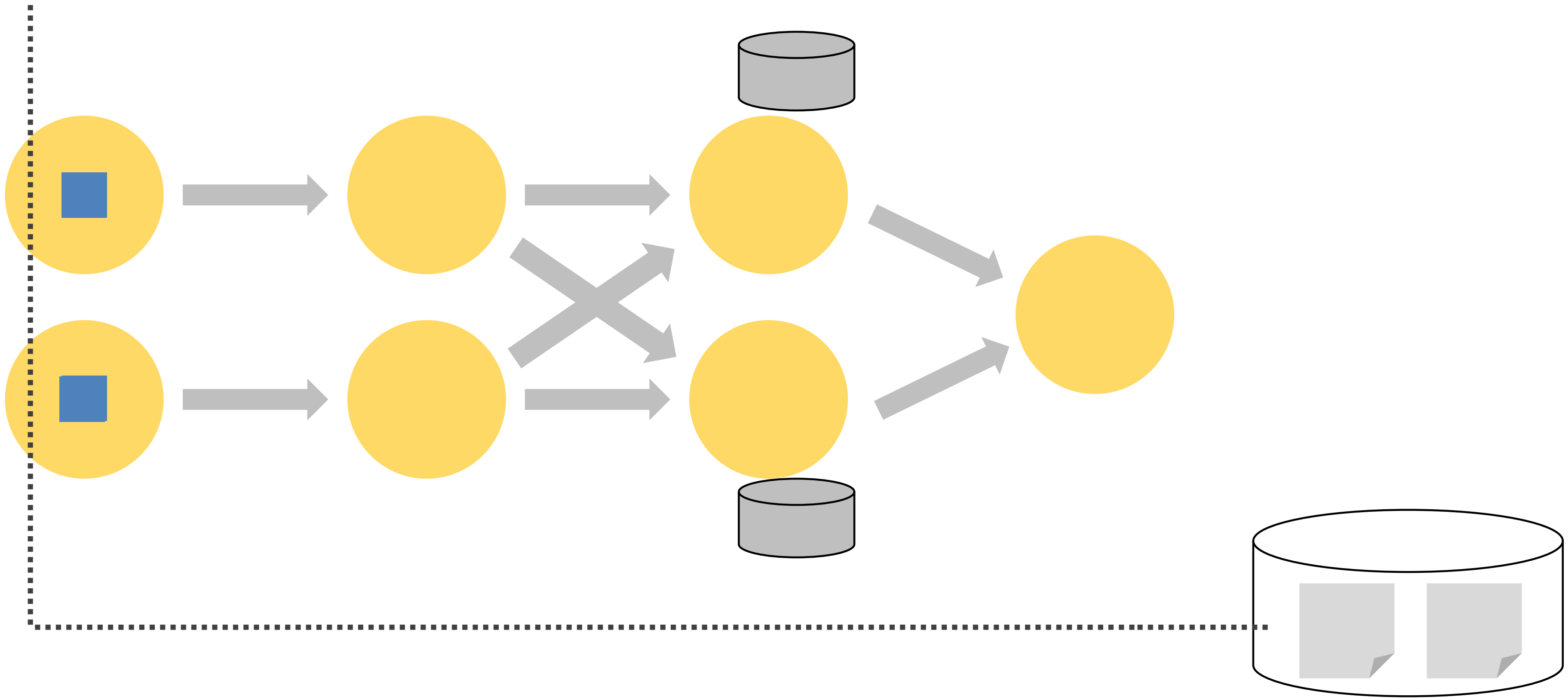


Event Sourcing + Memory Image

Recovery: Restore snapshot and replay events since snapshot



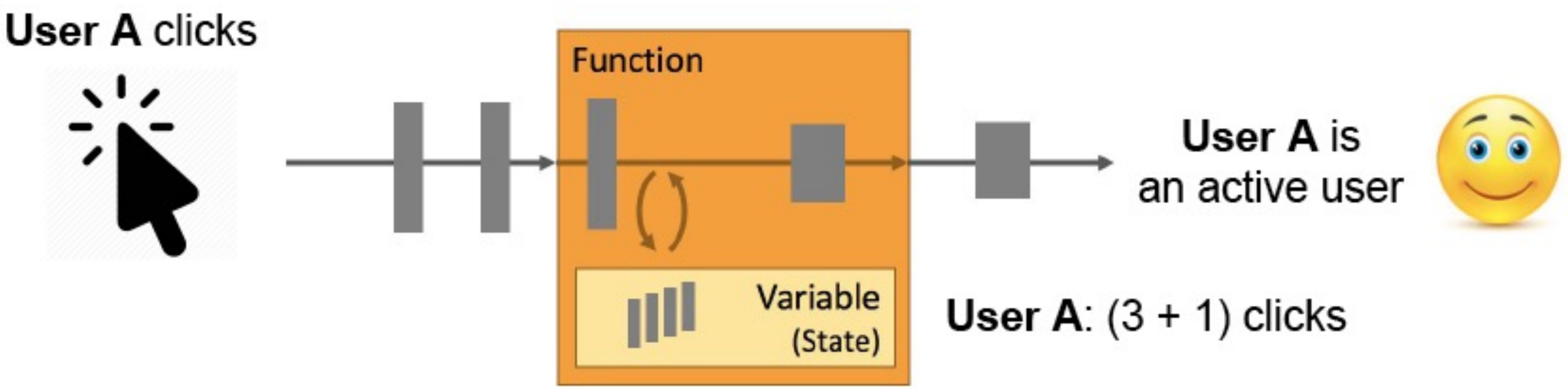
Stateful Event & Stream Processing



Checkpointing & Recovery

What is State in a Streaming Application ?

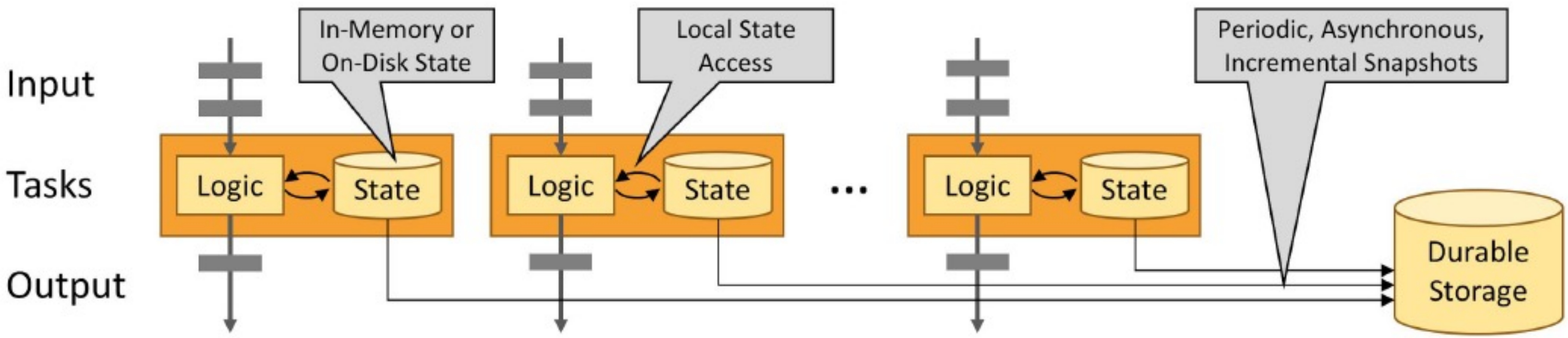
- Functions need to remember records or temporary results



- State is either per operator instance or per record key
- State backends: JVM heap or RocksDB on local disk
- Local == fast access

Maintaining and Checkpointing State

- State is periodically checkpointed to durable storage
  - A checkpoint is a <u>consistent</u> snapshot of the state of all operators

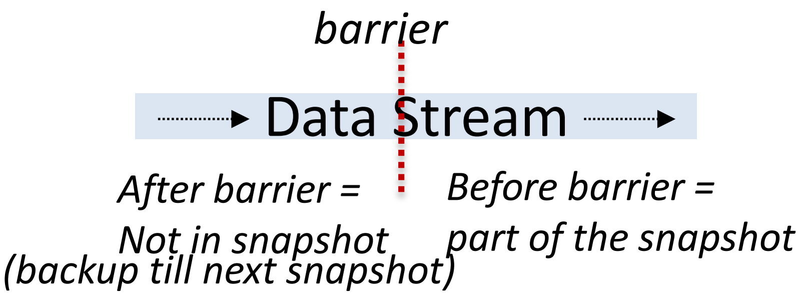


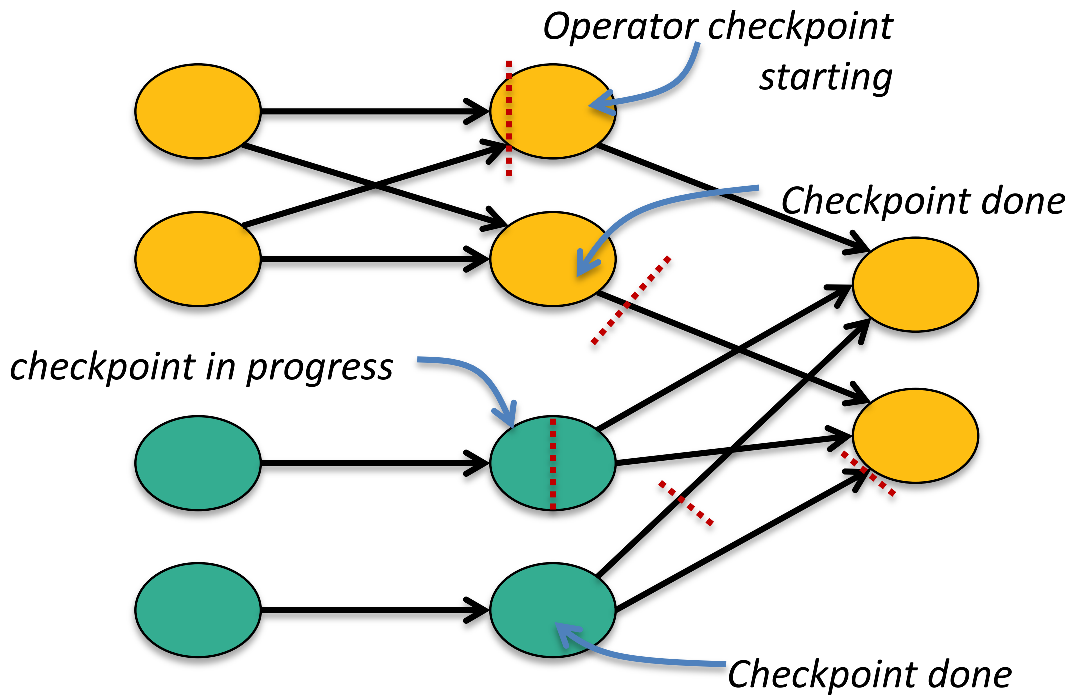
Checkpointing / Recovery

- Flink acknowledges batches of records
  - Less overhead in failure-free case
  - Currently tied to fault tolerant data sources (e.g., Kafka)
- Flink operators can keep state
  - State is checkpointed
  - Checkpointing and record acks go together
- Exactly one semantics for state

Checkpointing / Recovery

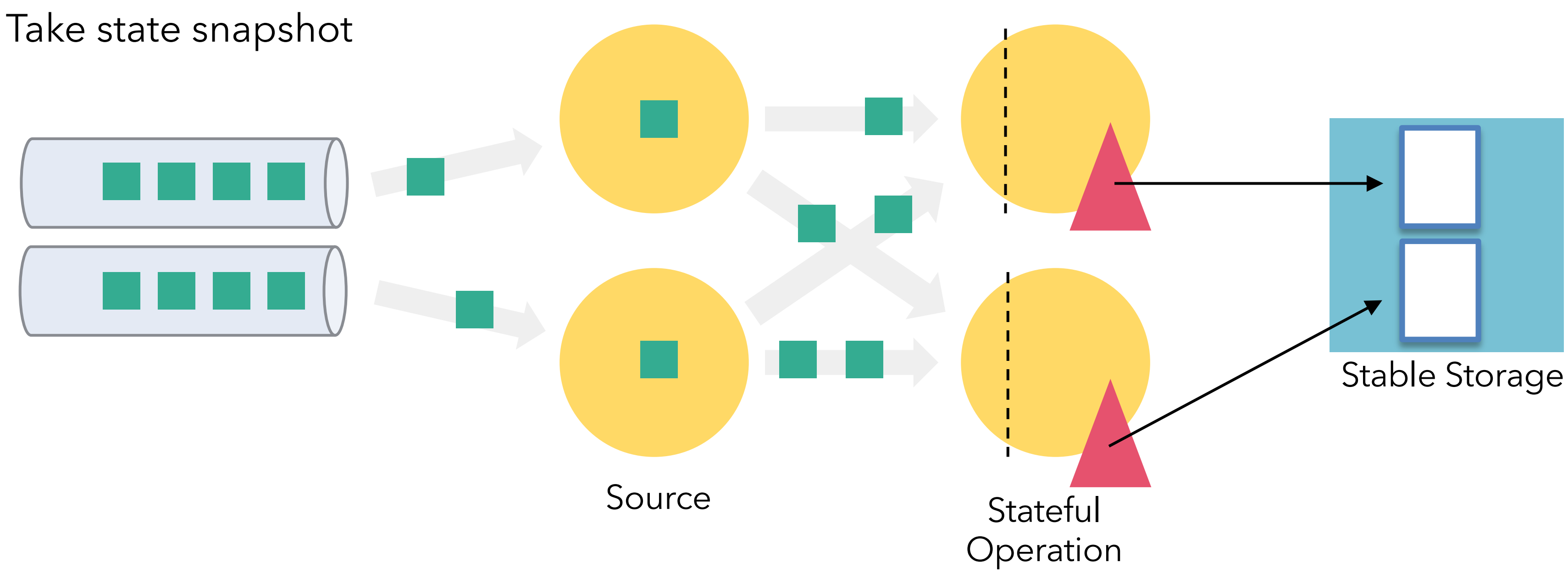
Pushes checkpoint barriers through the data flow





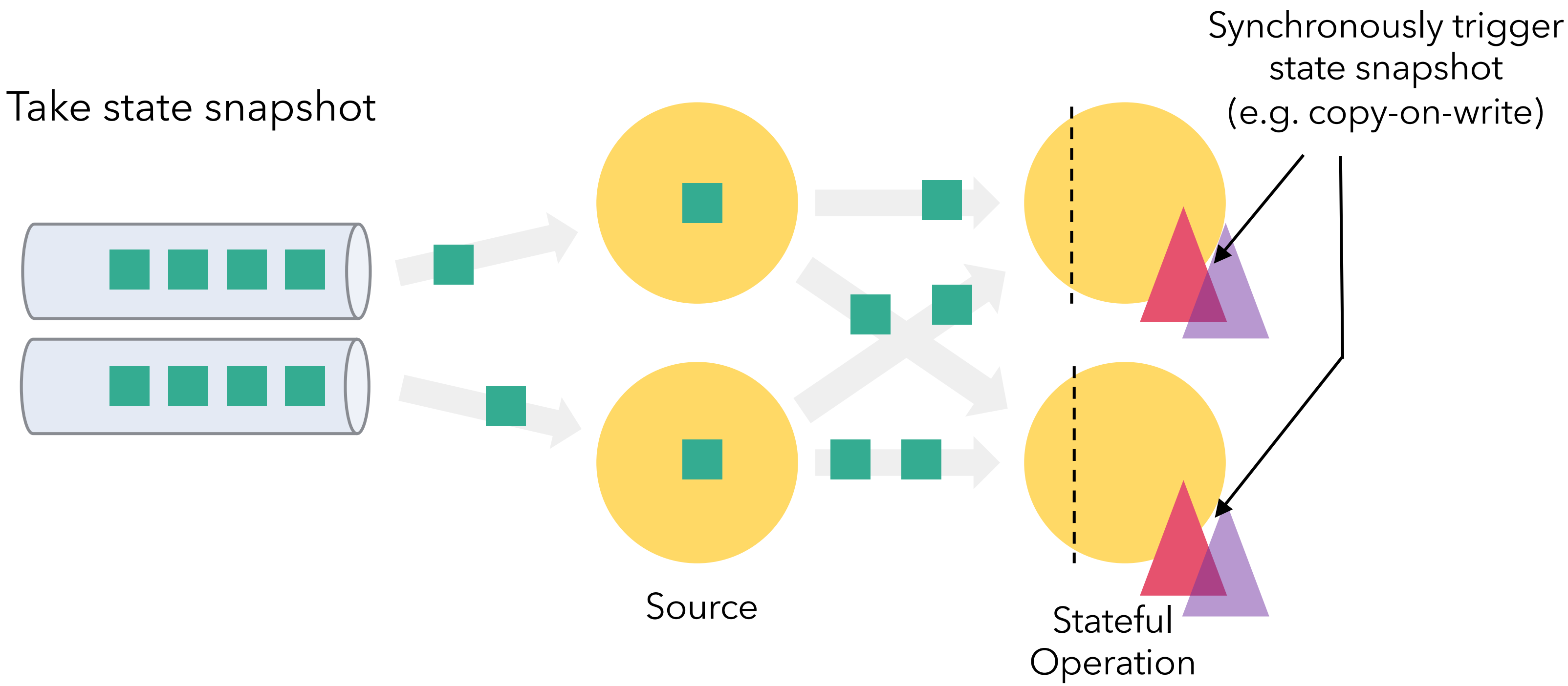
Chandy-Lamport Algorithm for consistent asynchronous distributed snapshots

Flink State and Distributed Snapshots

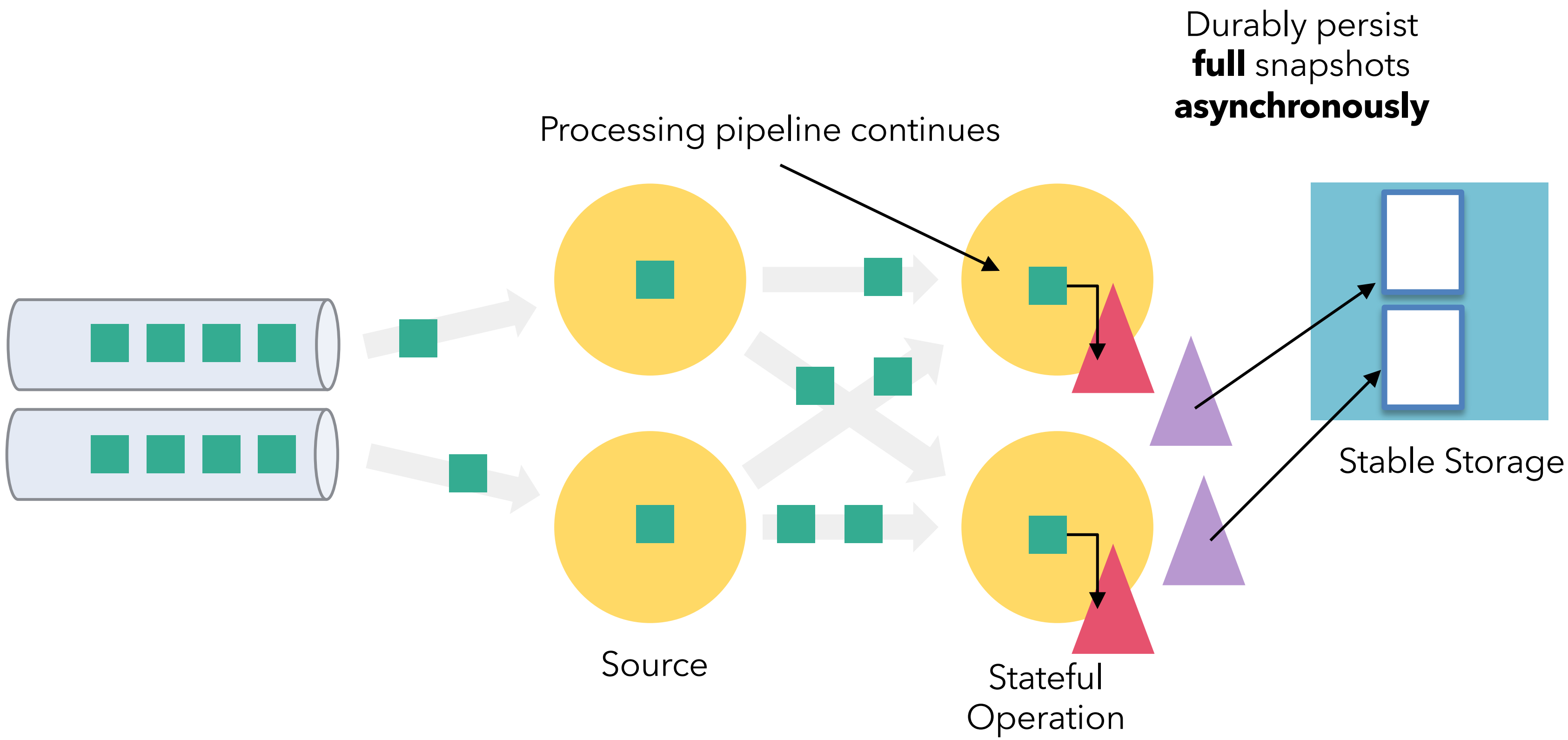


"Asynchronous Barrier Snapshotting"

Flink State and Distributed Snapshots

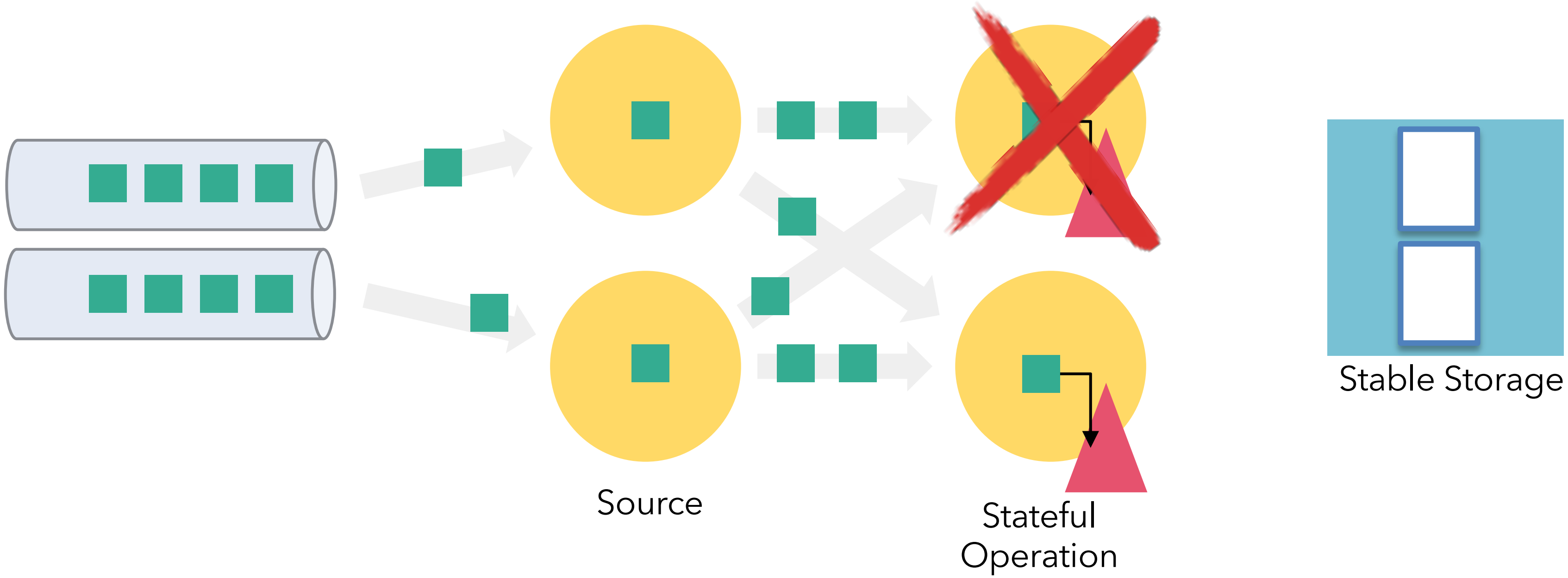


Flink State and Distributed Snapshots

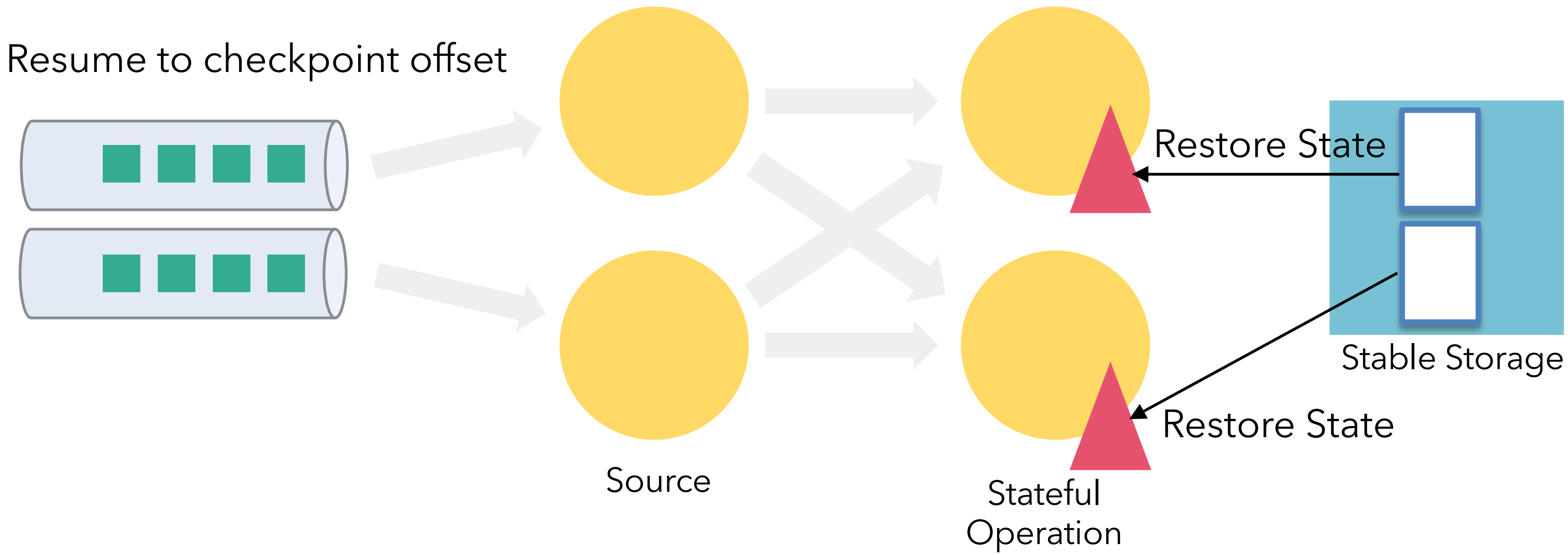


Task Local Recovery

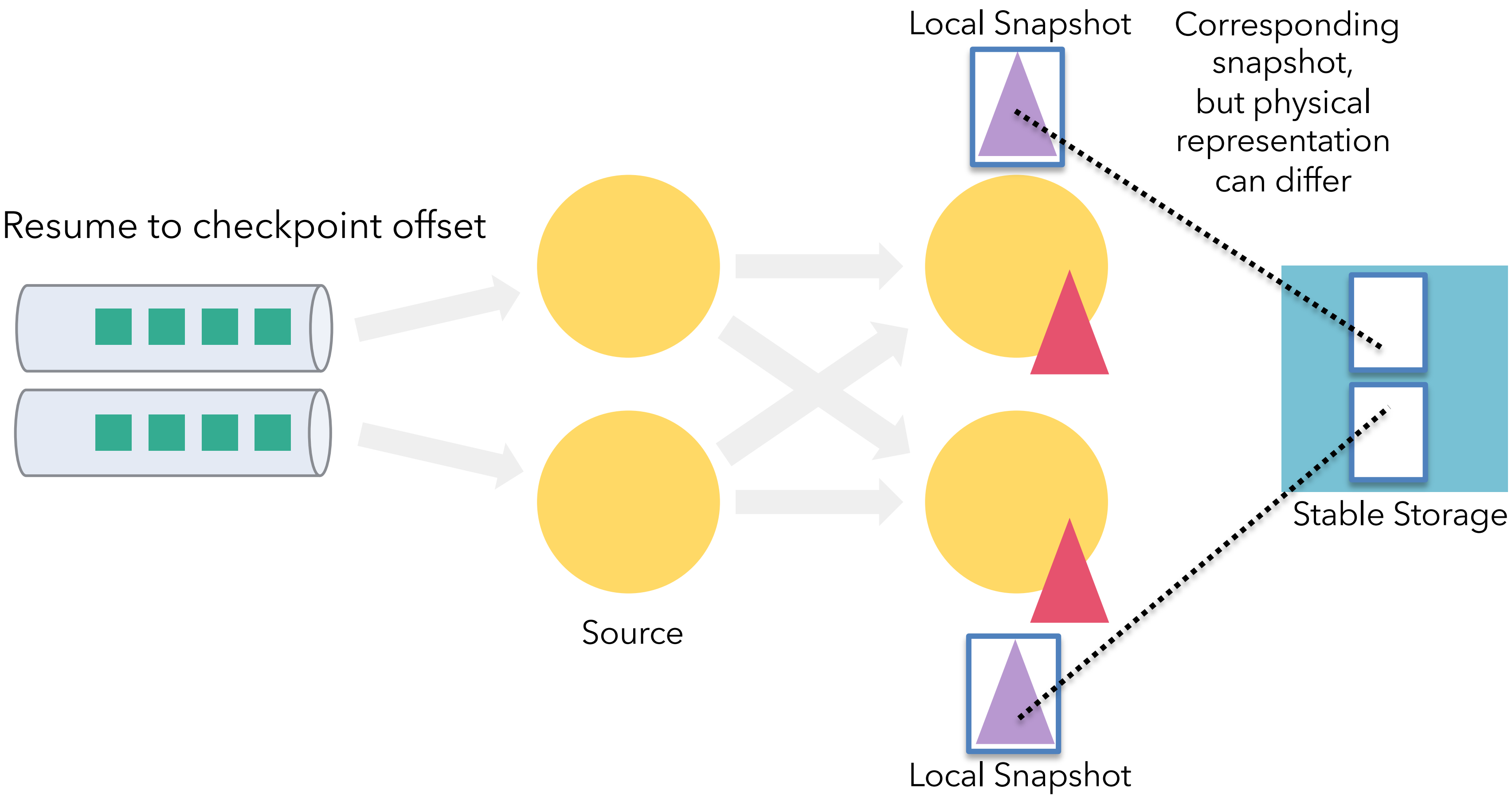
Recovery From Failure



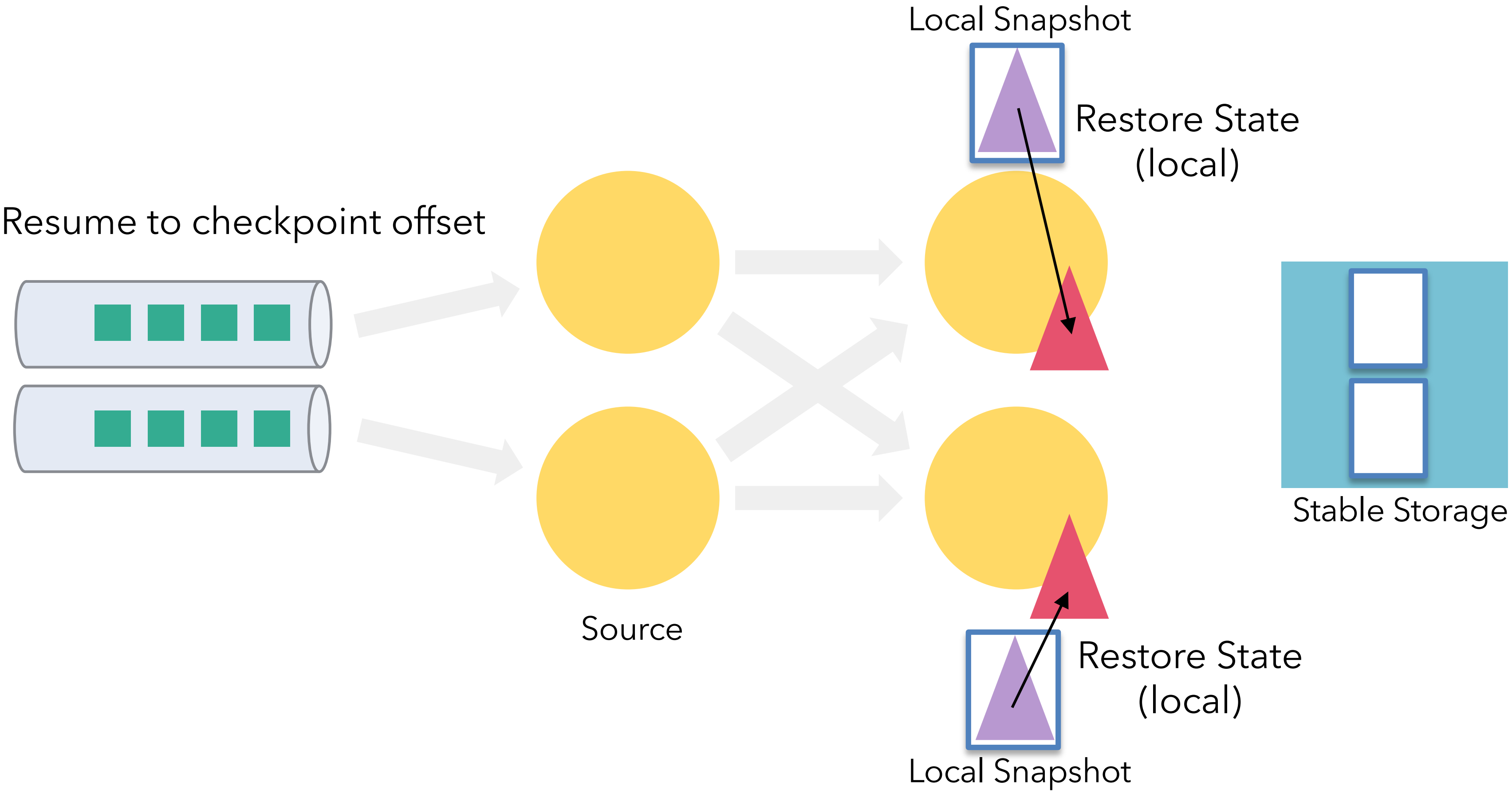
Recovery From Failure



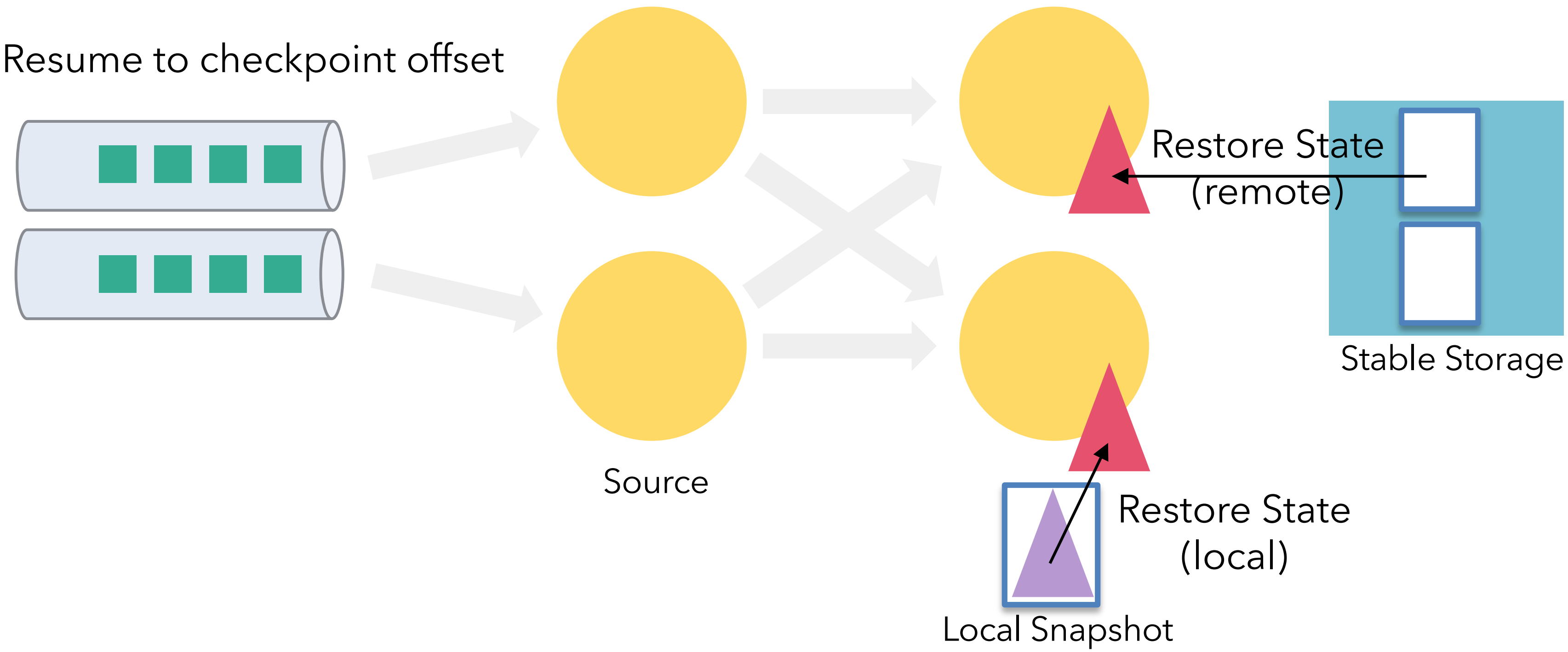
Local Recovery (Flink 1.5)



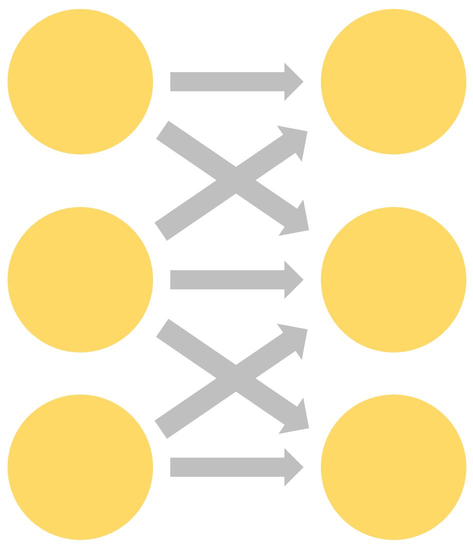
Local Recovery (TM survived)



Local Recovery (TM lost)

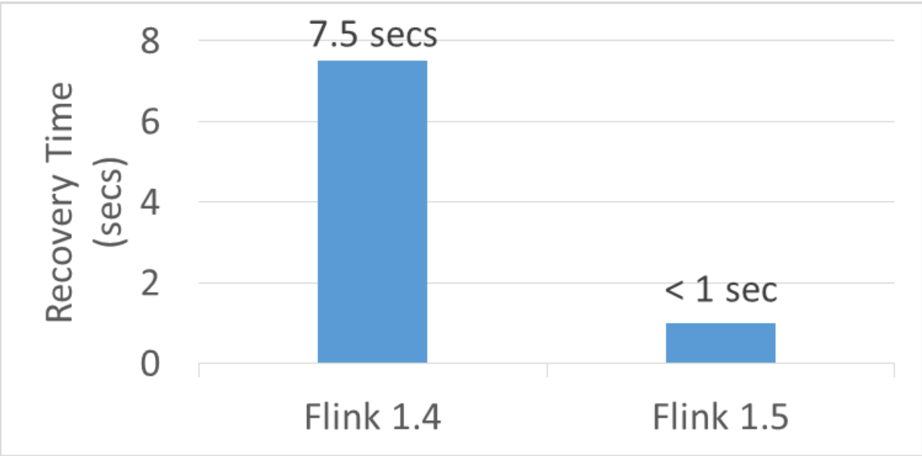


Localized State Recovery (since Flink 1.5)



Piggybags on internal Multi-version data structures:

- LSM Tree (RocksDB)
- MV Hashtable (Fs / Mem State Backend)

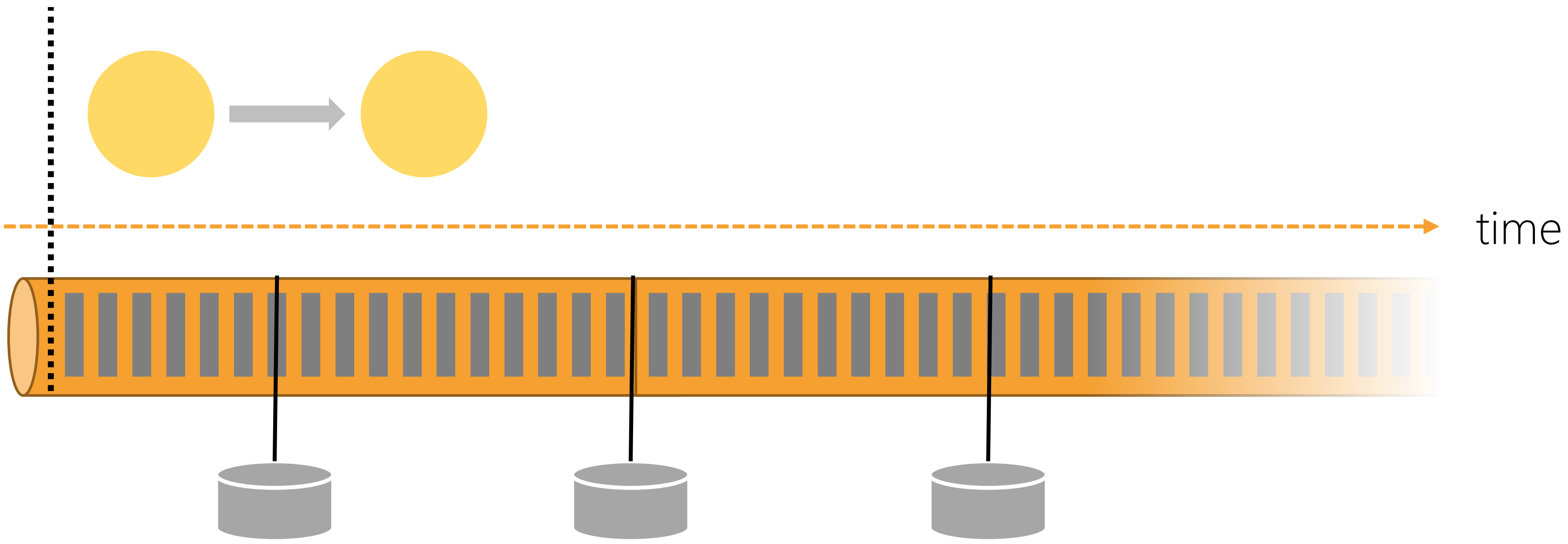


Setup:

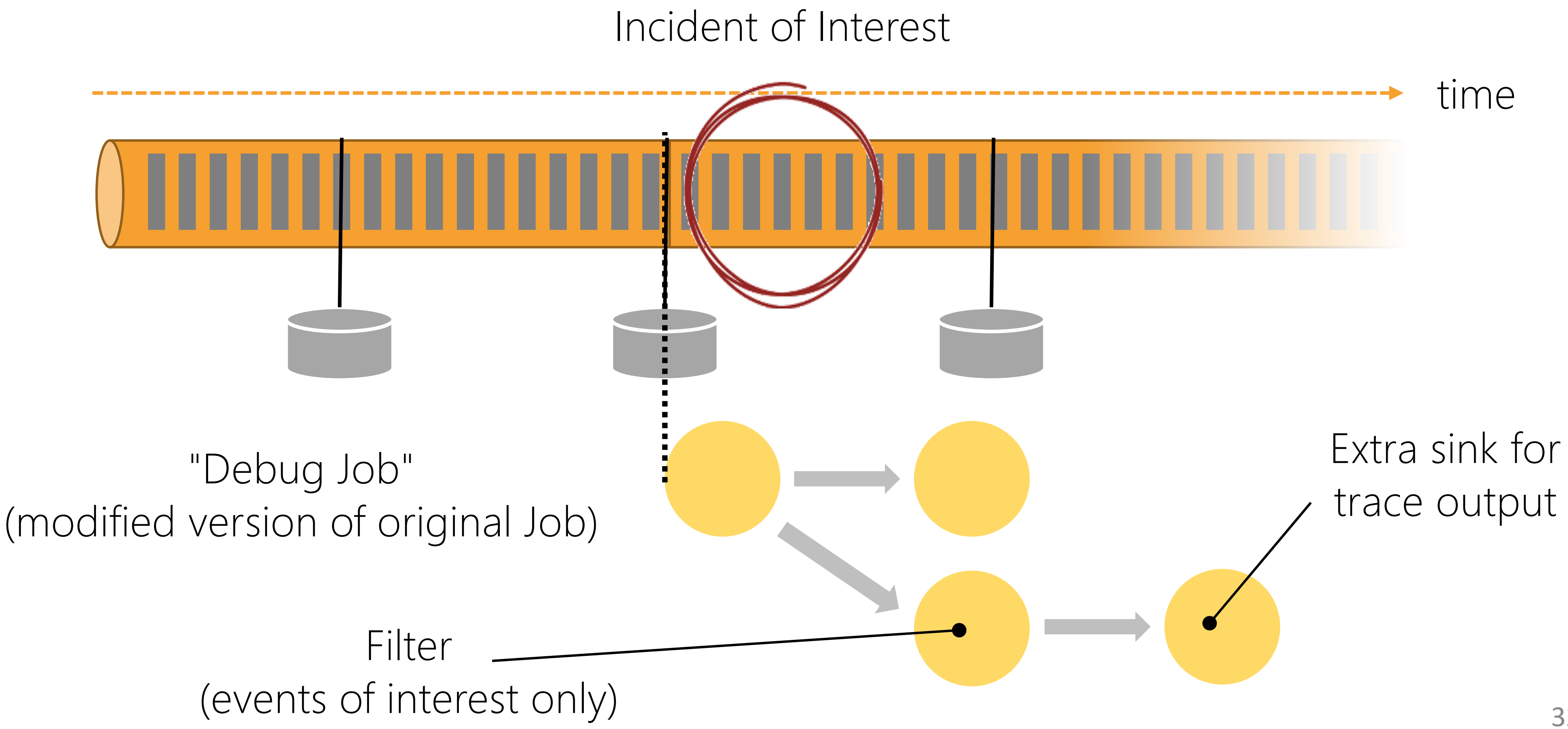
- 500 MB state per node
- Checkpoints to S3
- Soft failure (Flink fails, machine survives)

Having fun
with snapshots

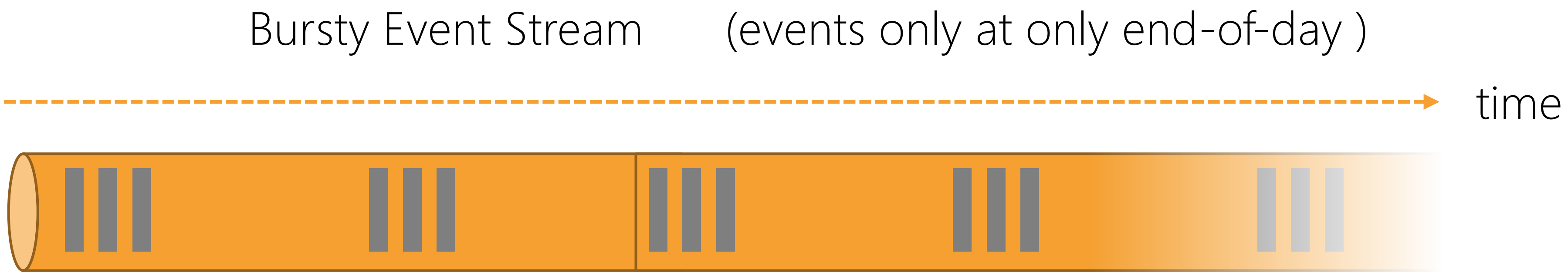
Creating periodic Snapshots



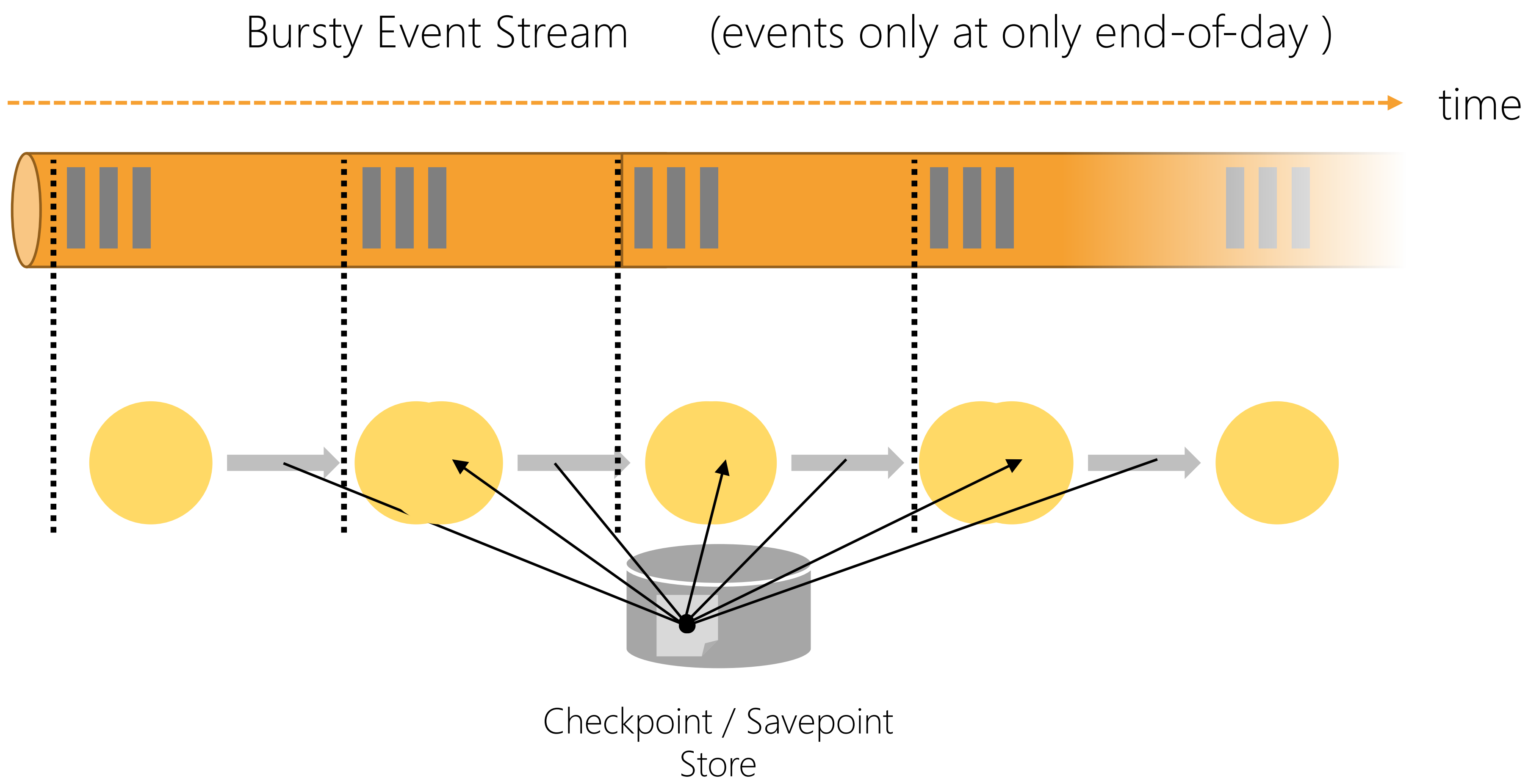
Replay from Savepoints to Drill Down



Pause / Resume style execution



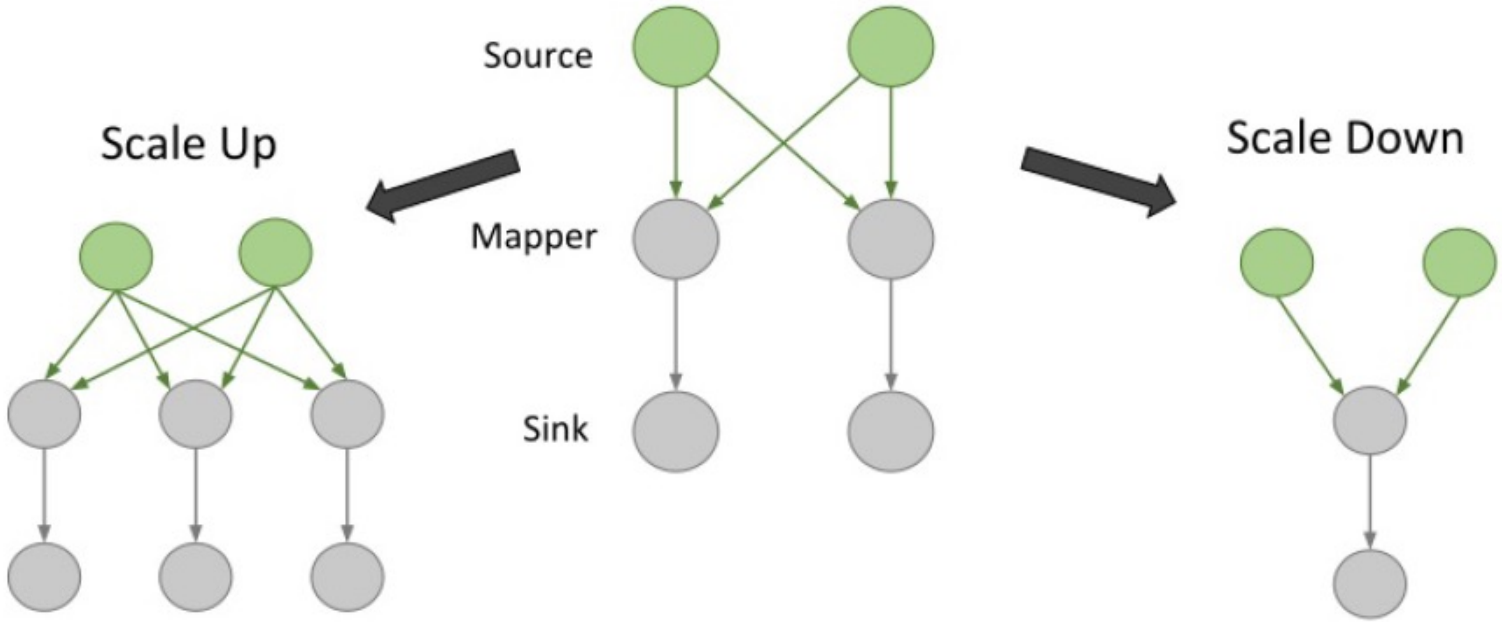
Pause / Resume style execution



Resource Elastisticity

Dynamic Scaling Flink applications

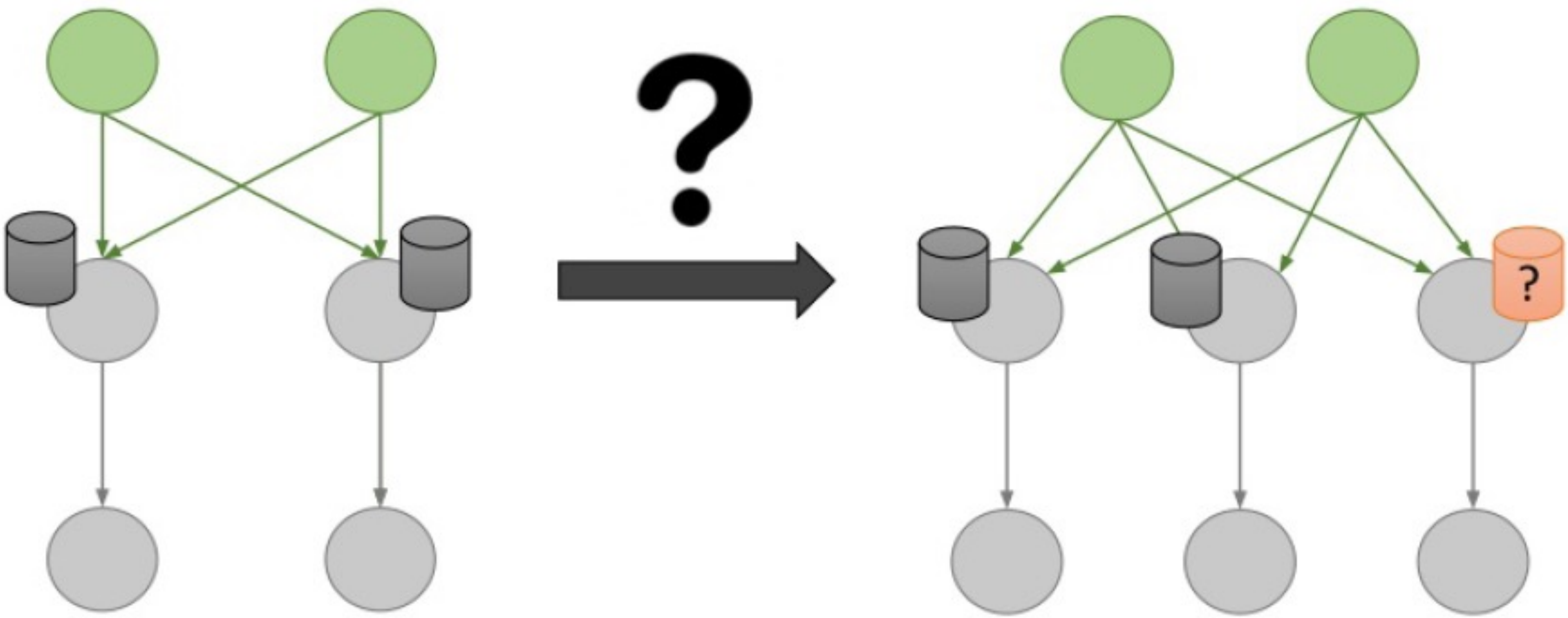
- Relatively Straightforward to Scale Stateless Jobs:



- Scale up: Deploy new tasks
- Scale down: Cancel running tasks

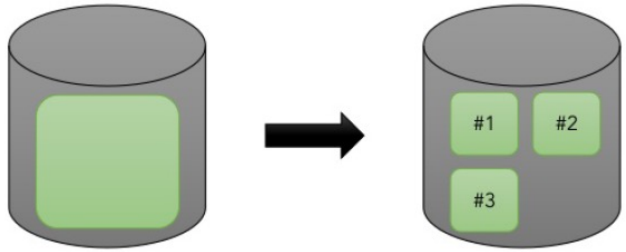
Dynamic Scaling Stateful Flink applications

- Problem: Which State(s) to assign to new task(s) ?

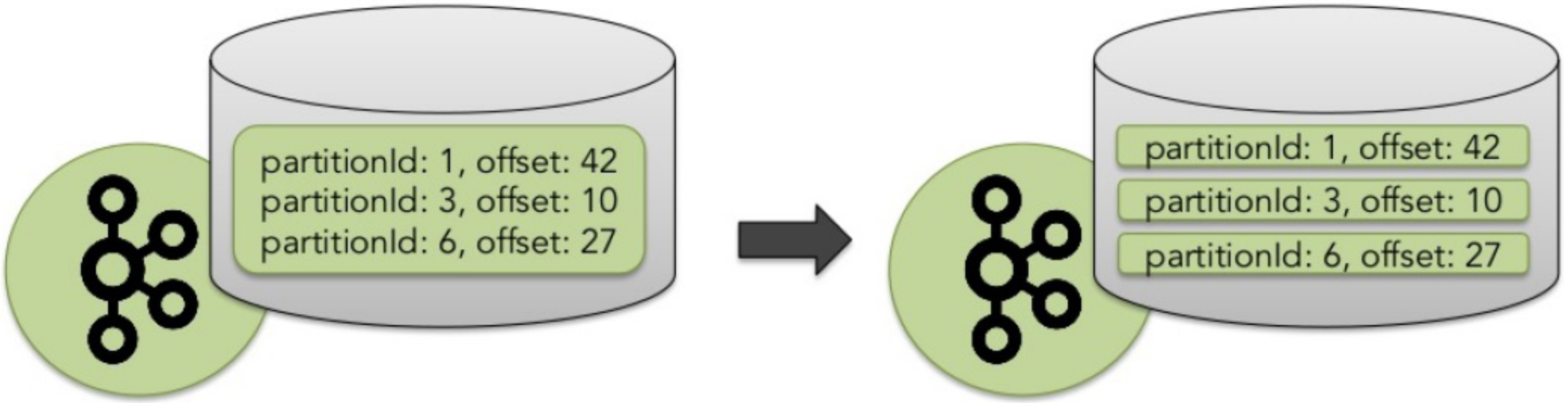


Repartitioning of Operator States

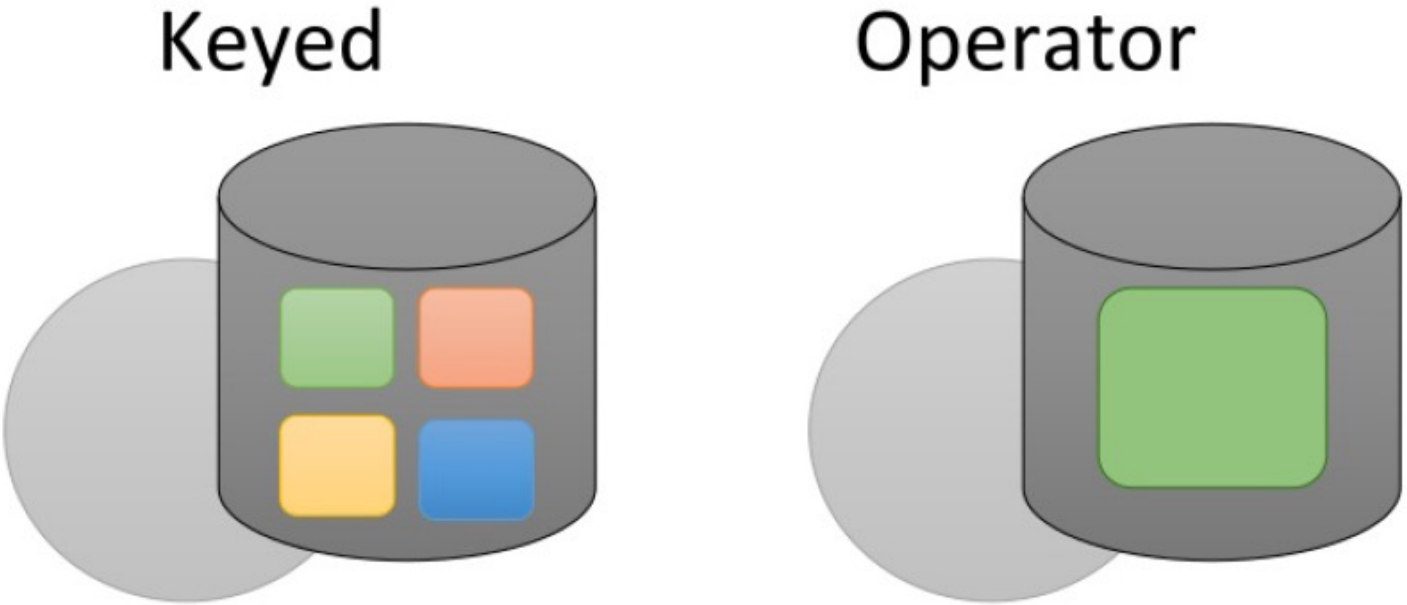
- Breaking Operator States up into Finer Granularity
  - State has to contain multiple entries
  - Automatic re-partitioning w.r.t. granularity



- Example: Kafka Source
  - Store Offset for each Partition
  - Individual entries are repartitionable



Keyed vs. Operator State

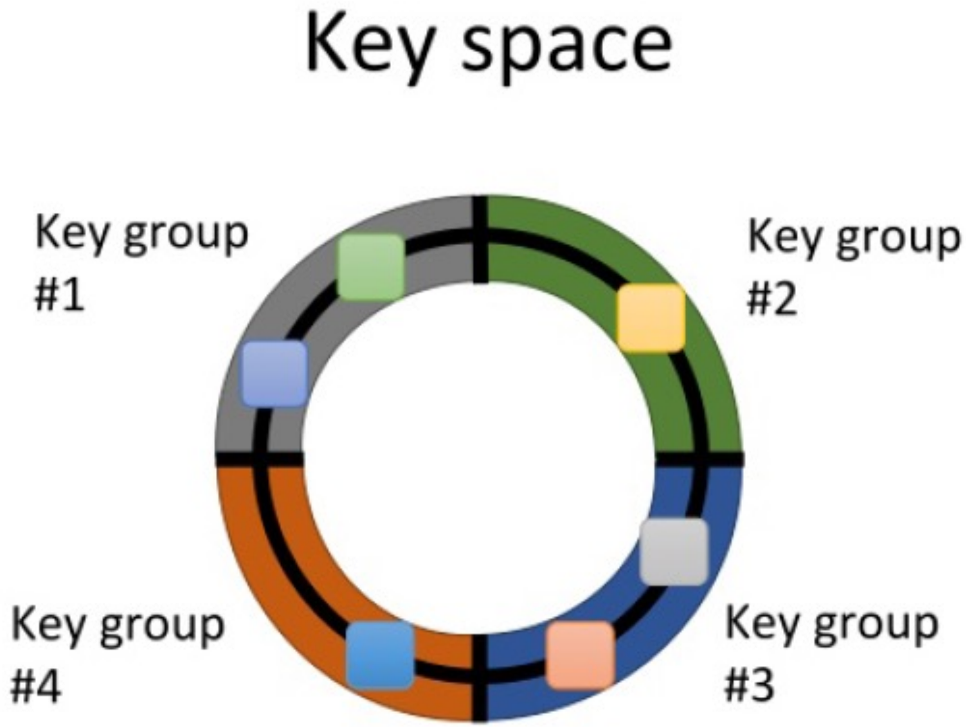


- State bound to a key
- E.g. Keyed UDF and window state

- State bound to a subtask
- E.g. Source state

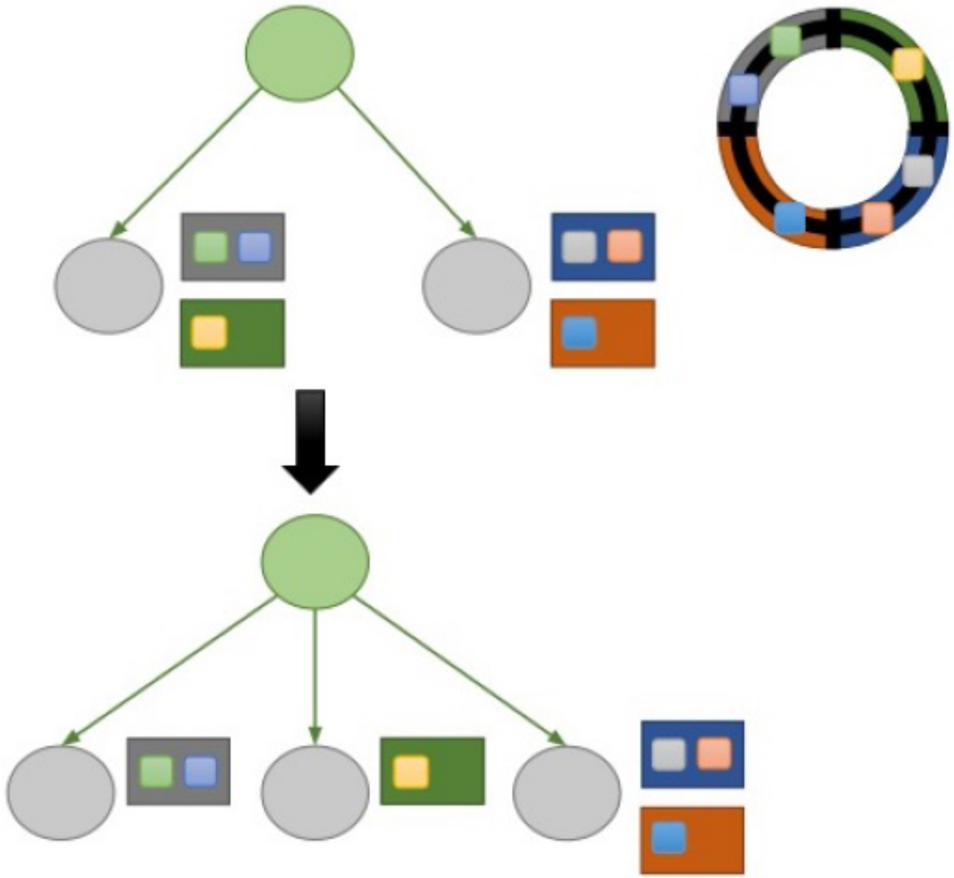
Repartitioning of Keyed States

- Similar to consistent hashing
- Split key space into key groups
- Assign key groups to tasks

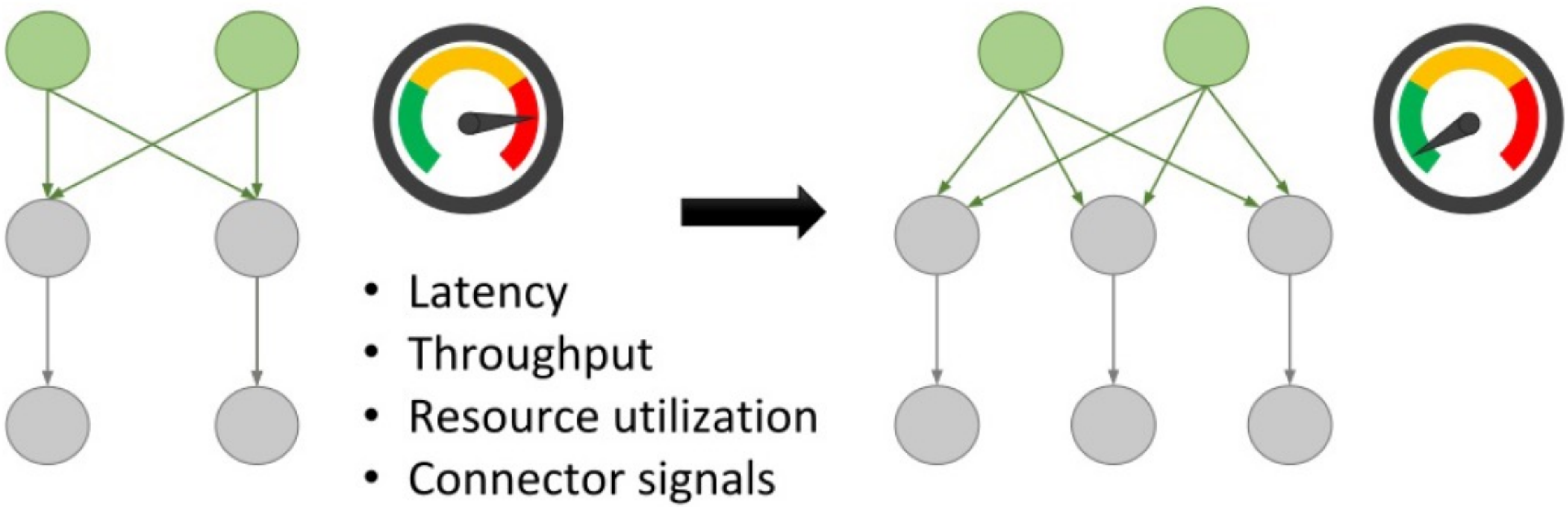


Repartitioning of Keyed States (cont'd)

- Rescaling changes key group assignment
- Maximum parallelism defined by #key groups



Automatic Scaling

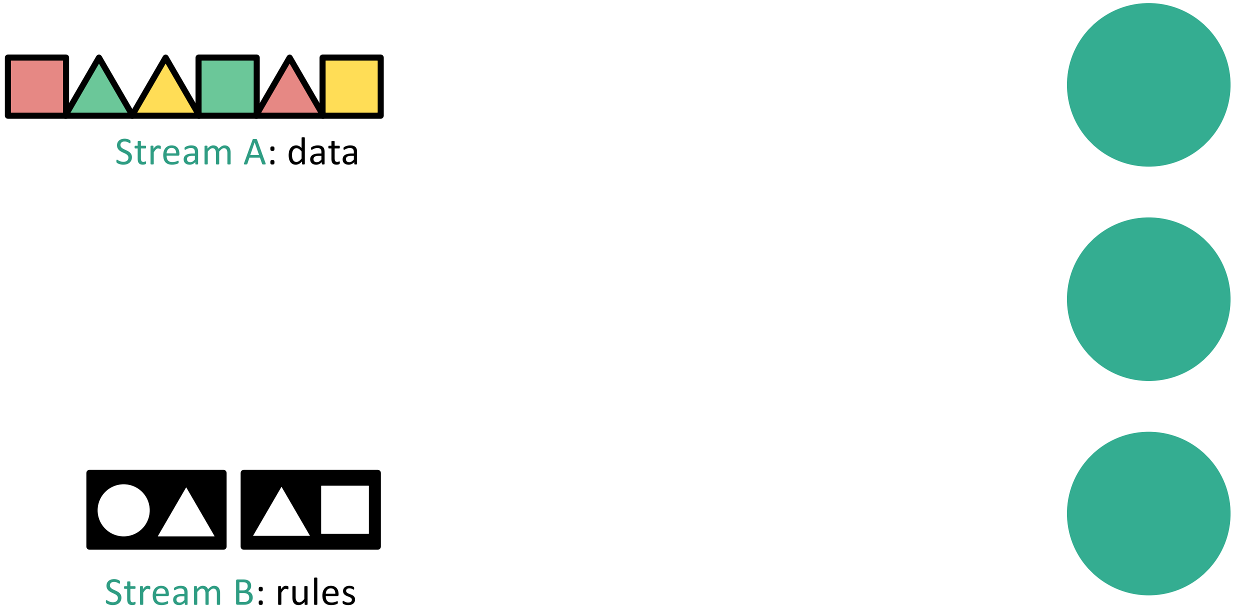


Broadcast State

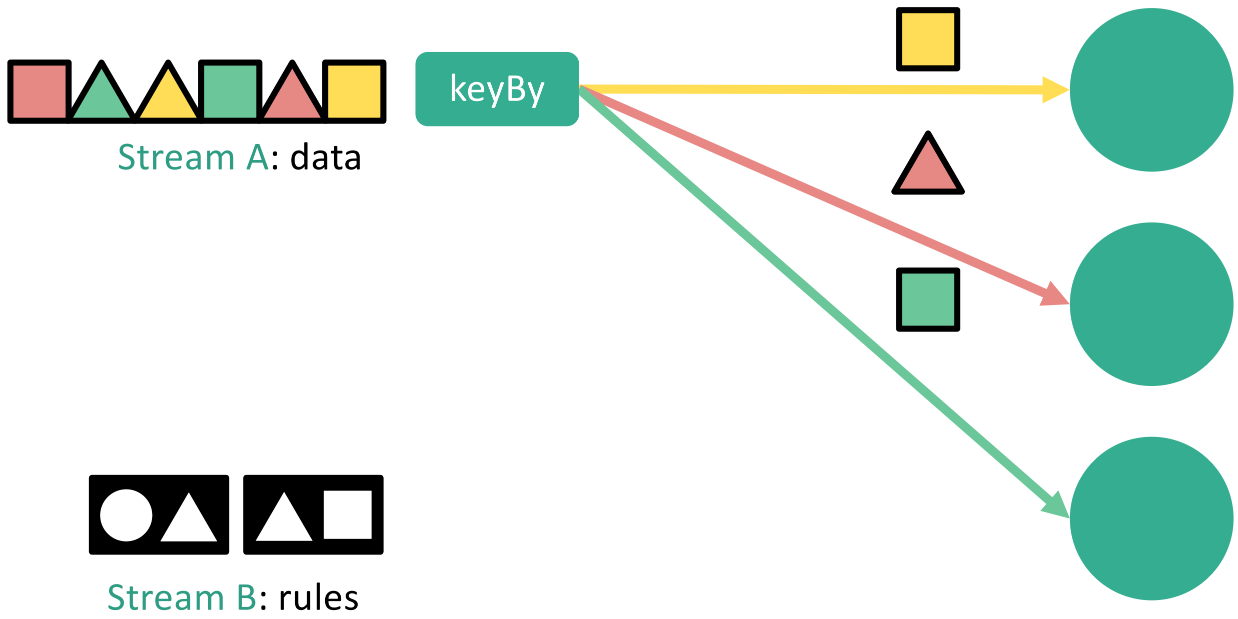
Why Broadcast State?

Evaluate a global, changing **Set of Rules** over a **(non-) keyed stream** of events.

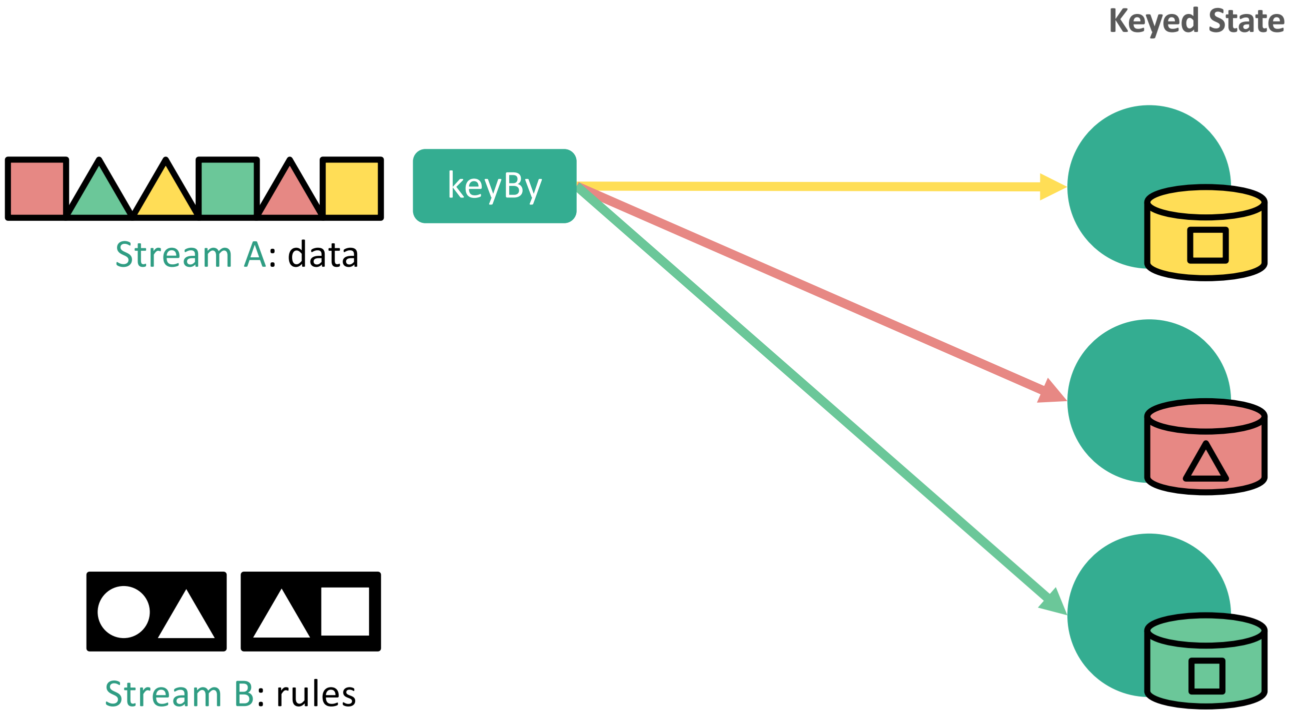
How to use Broadcast State



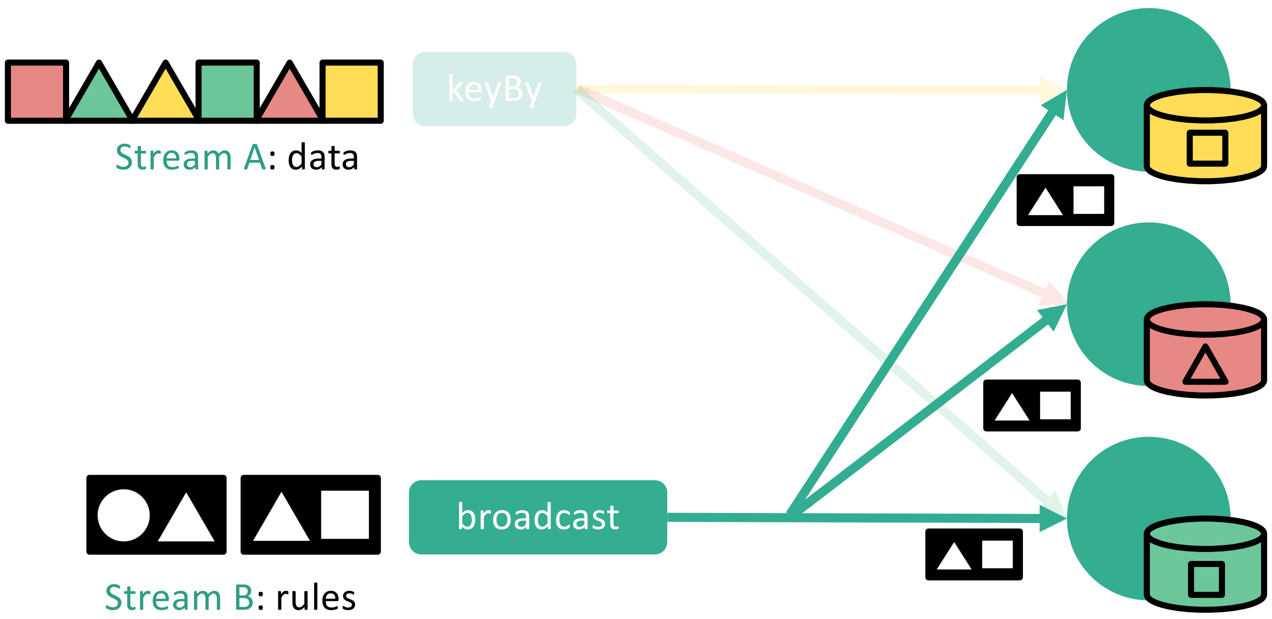
How to use Broadcast State



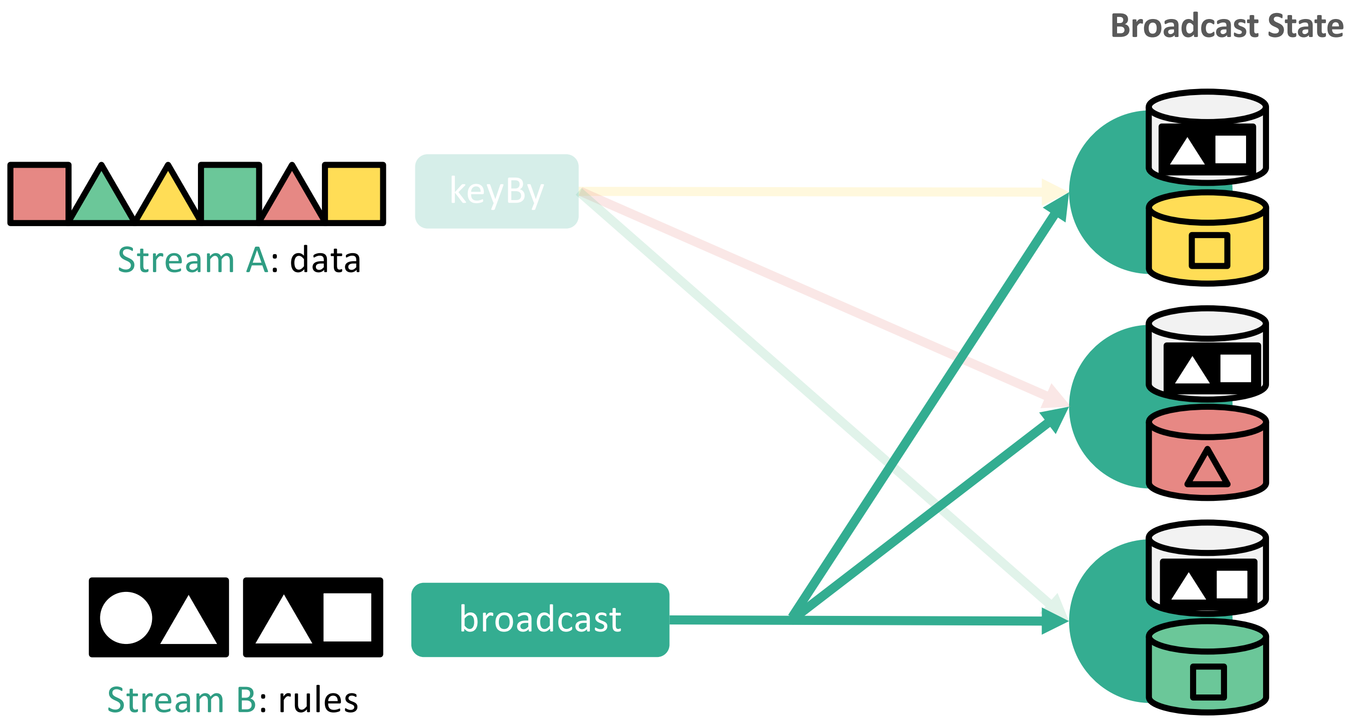
How to use Broadcast State



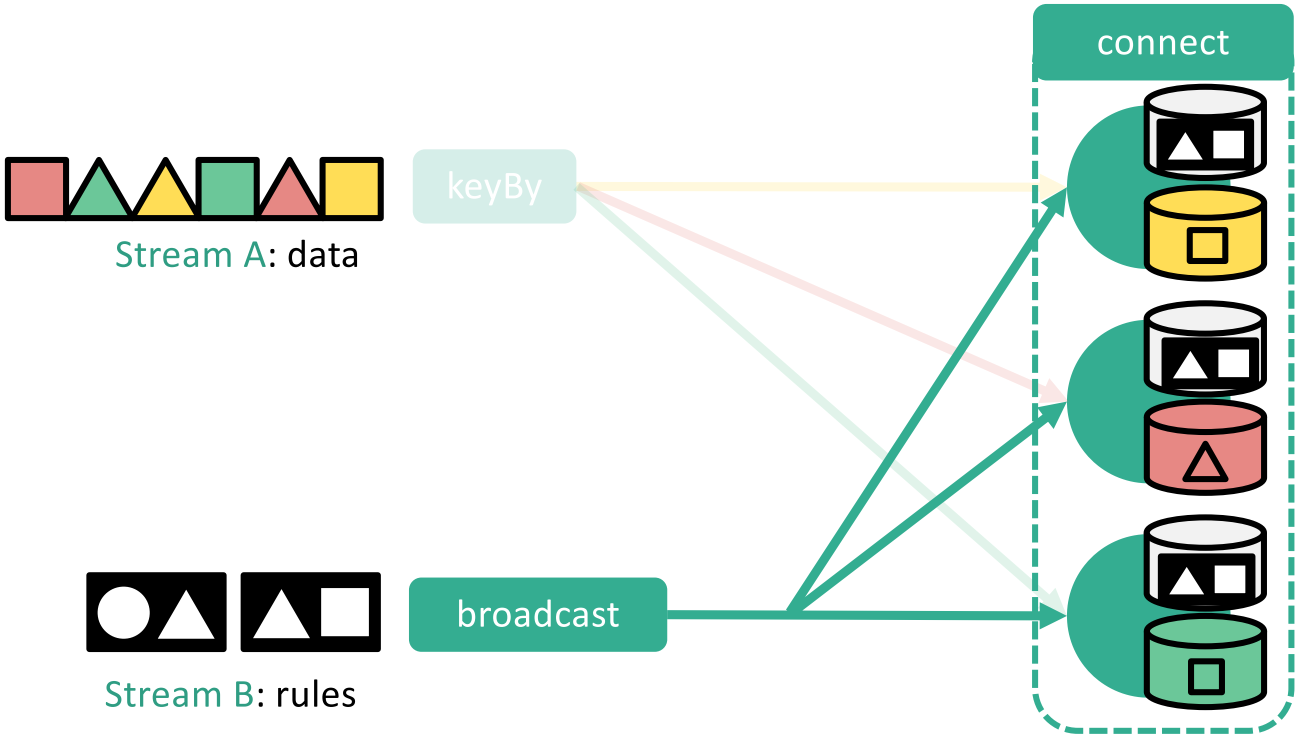
How to use Broadcast State



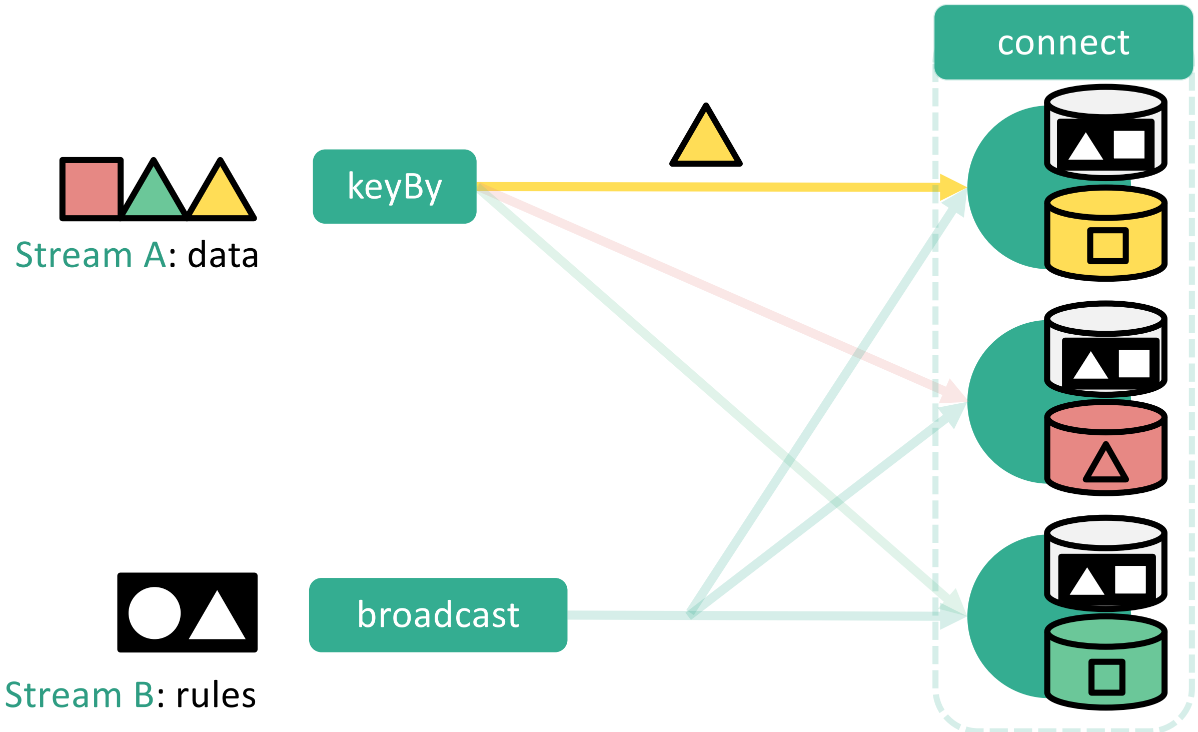
How to use Broadcast State



How to use Broadcast State



How to use Broadcast State



Broadcast State Wrap up

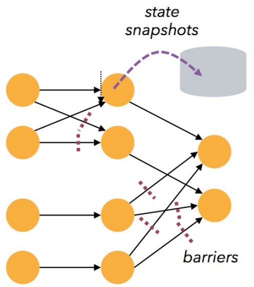
- Partition elements by key
- State associated to a key
- Broadcast elements
- State to store the broadcasted elements
  - Non-keyed
  - Identical on all tasks even after restoring/rescaling
- Ability to connect the two streams and react to incoming elements
  - Connect keyed with non-keyed stream
  - Have access to respective states

https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/state/broadcast_state.html

Backup/ Excess

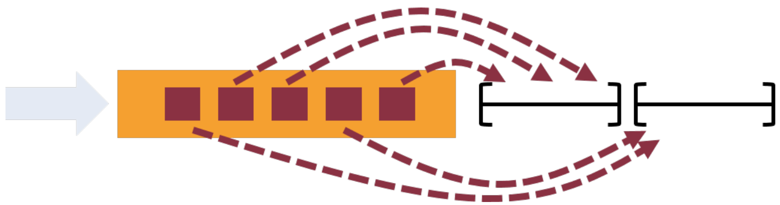
1. Failures and downtime

- Checkpoints & savepoints
- Exactly-once guarantees



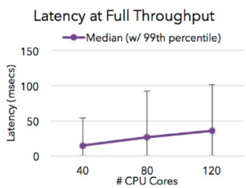
2. Out of order and late data

- Event time support
- Watermarks



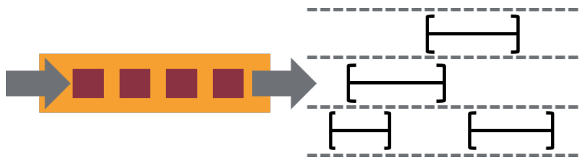
3. Results when you need them

- Low latency
- Triggers



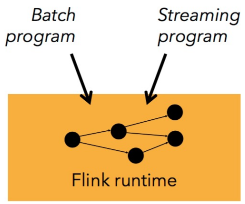
4. Accurate modeling

- True streaming engine
- Sessions and flexible windows



5. Batch + streaming

- One engine
- Dedicated APIs



6. Reprocessing

- High throughput, event time support, and savepoints

```
flink -s <savepoint> <job>
```

7. Ecosystem

- Rich connector ecosystem and 3rd party packages

8. Community support

- One of the most active projects with over 200 contributors

Summary: Cornerpoints of Flink Design

Flexible Data Streaming Engine

→ *Low Latency Stream Proc.*
→ *Highly flexible windowing semantics (i.e. think Beam)*

High-level APIs, beyond key/value pairs

→ *Java, Scala, Python(beta only)*
→ *Relational-style optimizer*

Additional Library Support

→ *Storm Compatibility Library*
→ *Graphs / ML Pipelines*
→ *ML & Streaming ML (catching up)*

Robust Algorithms on Managed Memory

→ *No OutOfMemory Errors*
→ *Scales to very large JVMs*
→ *Efficient Checkpointing/ Recovery & Saved points Op.*

Pipelined Execution of Batch Programs

→ *Better shuffle performance*
→ *Scales to very large groups*

Native Iterations

→ *Very fast Graph Processing*
→ *Stateful Iterations for ML*

What is Flink's unique contribution in the streaming data ecosystem?

Before Flink, users had to make hard choices between volume, latency, and accuracy

Flink eliminates these tradeoffs

- 10s of millions events per second for stateful applications
- Sub-second latency, as low as single-digit milliseconds
- Accurate computation results

A broader definition of accuracy: ***the results that I want when I want them***

1. Accurate under failures and downtime
2. Accurate under out of order data
3. Results when you need them
4. Accurate modeling of the world

Having a dependable framework enables more stateful applications to run as streaming applications